FBL: Part 2

# Boot Manager

**Overview**

This chapter focuses on the Boot Manager. You will learn about the following topics:

* Memory layouts
* Simplified startup
* Detailed startup
* Structures
* Configuration of standalone Boot Manager
* Failsafe Updater

**Boot Manager (BM)**

The Boot Manager (BM) controls the booting process of the controller. It decides whether to start targets like FBL, application, FBL Updater or the Failsafe Updater. The BM enables users a great flexibility of user configuration and user implementation e.g. initialization, target invocation and more.  
The BM uses bus communication only in rare cases. Though, communication to external memories for example through SPI is possible.

**Validation Information**

The BM can read the target’s validation information. Usually, pre-implementations for presence patterns (PP) and NV-Memory are available. Users can implement their own validation access methods. Important is that the BM needs to access the validation information in the same manner as the FBL stores the validity information. For example the FBL writes to application validity presence pattern to the last downloaded logical block then the BM must check the logical blocks for a valid application validity presence pattern.

**Tips**

When validity information is stored on external memory devices or through memory abstraction components like FEE/EA, the according components and drivers must be integrated into the BM.

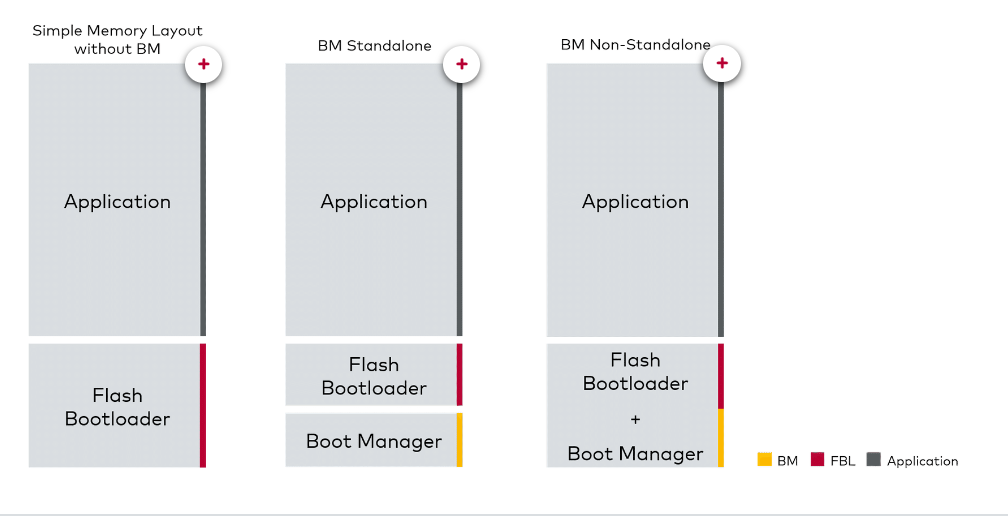
**FBL Updater**  
If the FBL Updater is part of the ECU software, the BM ensures hardware independent reset safe updates. Meaning the BM ensures such updates, even when the hardware's architecture won't allow reset safe updates. This is because the BM is not exchanged during FBL updates and always acts as first entry point when the hardware is powered on.

**Failsafe Updater**  
When the Failsafe Updater is enabled, the BM will first check whether a valid FBL is available. Afterwards  the BM checks if any other valid target is available on the ECU. If neither FBL nor any other target is valid, the BM will automatically search for the Failsafe Updater on the controller. Such a Failsafe Updater may be stored in the application’s area where the FBL Updater is usually stored. Nevertheless, the Failsafe Update can be stored in any flash blocks which are accessible and executable. The search algorithm of the BM uses an integrated flash block table since the FBL’s logical block table is not accessible anymore.

**Secure Boot**  
During secure boot startup procedures, the BM can act as hardware trust anchor (HTA) and triggers the verification of the software to be started. This is done by interacting with the Hardware Security Module (HSM) or any Secure Hardware Extension (SHE).

**Boot Manager Memory Layouts**

There are three different memory layouts possible. The “Simple Memory Layout without BM” which has been described earlier and “BM Standalone” and “BM Non-Standalone”.  
If a BM is available in the FBL’s SIP, the BM can be compiled either standalone (BM Standalone) or with the Flash Bootloader (BM Non-Standalone).

****

**Tips**

If the BM is part of the FBL’s SIP, the BM must be used since the FBL lost the ability to start a valid application.

**Boot Manager Startup**

**Boot Manager Simplified Startup**

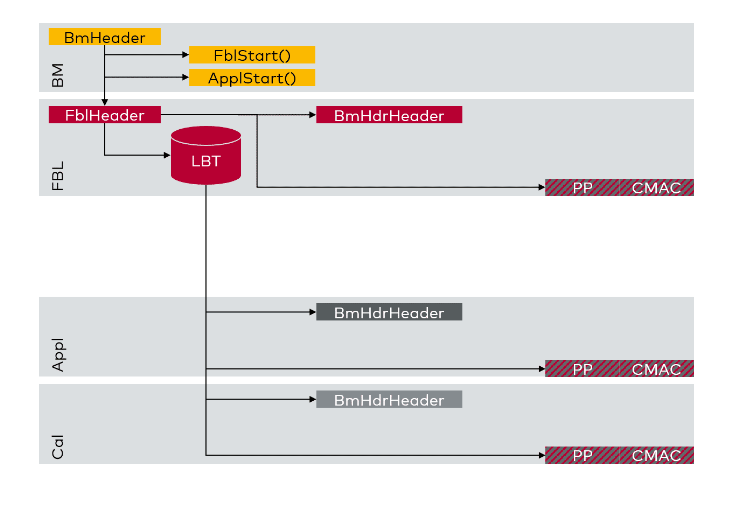
The BM, the FBL and any other boot target has its own startup code. This ensures that each program can operate in its own context. However, the startup timing can be improved when redundant executions like RAM initialization are only performed by one program.

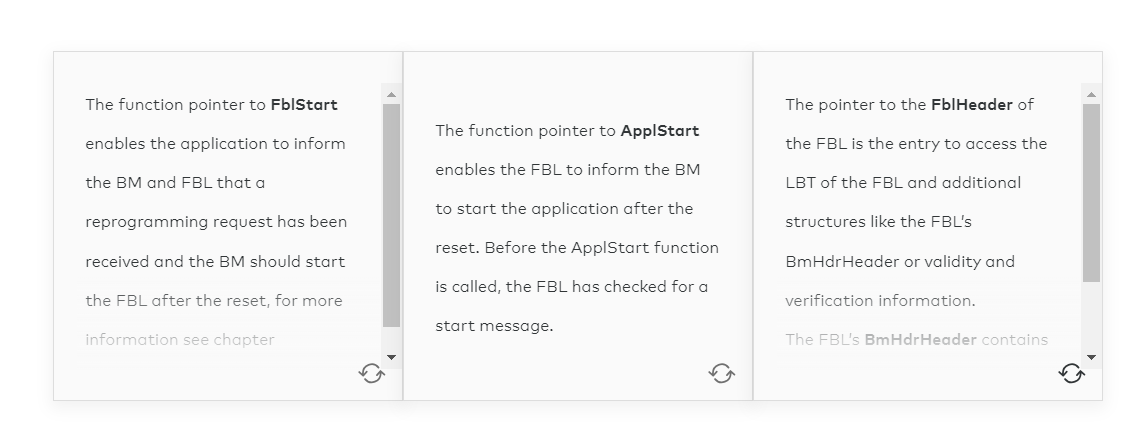
**Boot Manager Detailed Startup**

The startup procedure of the BM highly depends on features like ‘Stay in Boot’, ‘Secure Boot’ or ‘Failsafe Updater’. Though the startup behavior won’t change for standalone or non-standalone configurations.

**Boot Manager Structures**

The BM accesses different structures in order to decide which boot target is allowed to be started. The BM itself contains the structure BmHeader. This structure has different elements like the version information of the BM, the pointer to the FblHeader in the FBL and function pointers to the FblStart and ApplStart function.



****

The FBL’s **BmHdrHeader** contains the FBL’s entry address and depending on the feature set the verification structure.

The access to all available boot targets is ensured through the LBT. It has information about the logical blocks, their BmHdrHeader address and depending on the configuration the presence pattern address is calculated. Meaning the access to the validity information of the logical block is possible through the LBT.

**Example**

In step 6 (Search Target) of the previous graphic which shows the detailed startup, the BM searches the bootable targets.

Starting from the BmHeader the BM will check the FblHeader, the LBT and then the validity information of the desired boot target. Is the boot target valid, the entry address which is stated in the BmHdrHeader is directly jumped to. From that point on the context is switched to the boot target which will start with its own startup sequence.

Whether the presence pattern is available for each logical block is configuration dependent. The same applies for the CMAC (Cipher Message Authenticated Code) which is usually only available for SHE-based Secure Boot configurations.

**FAQ**

Below we have compiled a few questions that are frequently asked. Maybe you can answer one or two of them yourself? The answer will be displayed when you click on the plus sign.

**How can the FblHeader be accessed by the BM?**

–

The BM’s BmHeader structure contains a direct link to the FblHeader. When the BM is compiled standalone the BmHeader will store the configured address of the FblHeader in the Bmheader. Since this configured address is hard coded in the BM’s configuration the FBL must ensure that the FblHeader is linked to the same address. E.g., the BM expects the FblHeader to be placed at address 0x2000 the FBL has to link its FblHeader to address 0x2000 as well.

**How can the LBT accessed by the BM?**

–

The BM will access the FBL’s LBT through the FblHeader. The FblHeader must be linked to the same address as configured in the BM. The FblHeader contains a pointer to the LBT.

**How can the BmHdrHeaders of the boot targets be accessed?**

–

The user must configure the BmHdrHeader in the LBT. For each logical block there is an address of the BmHdrHeader configured.

**How is the BmHdrHeader of the FBL accessed?**

–

Since the FBL is not considered in the LBT the BmHdrHeader of the FBL is referenced in the FblHeader and not in the LBT.

**Configuration of Standalone Boot Manager**

Some configuration parameters are shared between FBL and BM.  
These configuration parameters shall be configured equally in BM and FBL

* FblStart
* FBL Header Address
* Validity Handling e.g. presence pattern
* Reprogrammable LBT
* Stay In Boot (Force Boot Mode)
* XCP

Configuration parameters which shall be configured equally are Secure Boot (on/off, Key Initialization, MAC Initialization) and BM Standalone Mode.

Configuration parameters of FblBmHdrTargets must be equally configured among all configurations (BM, FBL, boot targets).

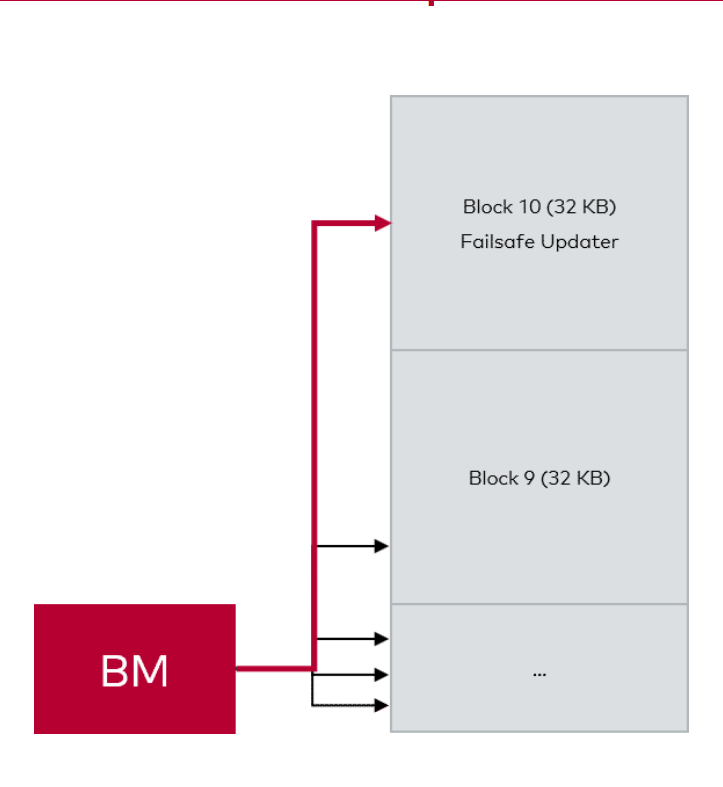
**Failsafe Updater**

BM searches for the Failsafe Updater if the FBL is invalid/unavailable. Each flash block in the BM's flash block table is checked for a BmHdrHeader and whether it is at a pre-implemented offset relative to the flash block's start address. The offset should be configured relative to the offset of the BmHdrHeader by the Application/FBL Updater. Nevertheless, the search algorithm can be customized by the user. The flash block must be assigned to a memory block. The BM is able to search for the Failsafe Updater in any memory block. The memory driver must be compiled with the BM. The Failsafe Updater must not be configured as a target.

**Example**

Failsafe Updater is located in block #10.  
Offset is configured to 0x4000 (16KB).  
BM checks

* each flash block‘s offset address
* for a BmHdrHeader
* containing the Updater‘s target ID



**ApplVect and Transitions**

*Lesson 2 of 9*

**Overview**

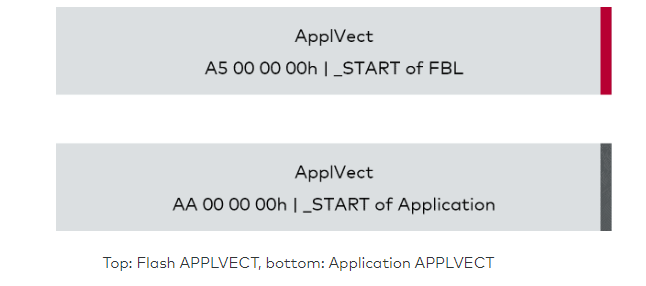
In this chapter you will learn about different transitions between the FBL and the application:

* Transition from FBL to application (without BM)
* Transition from BM to FBL, application or updater
* Transition from application to FBL

But before this you will learn about the Application Vector Table (ApplVect), since it is needed for the start of the application in non-BM based FBLs.

**Application Vector Table**

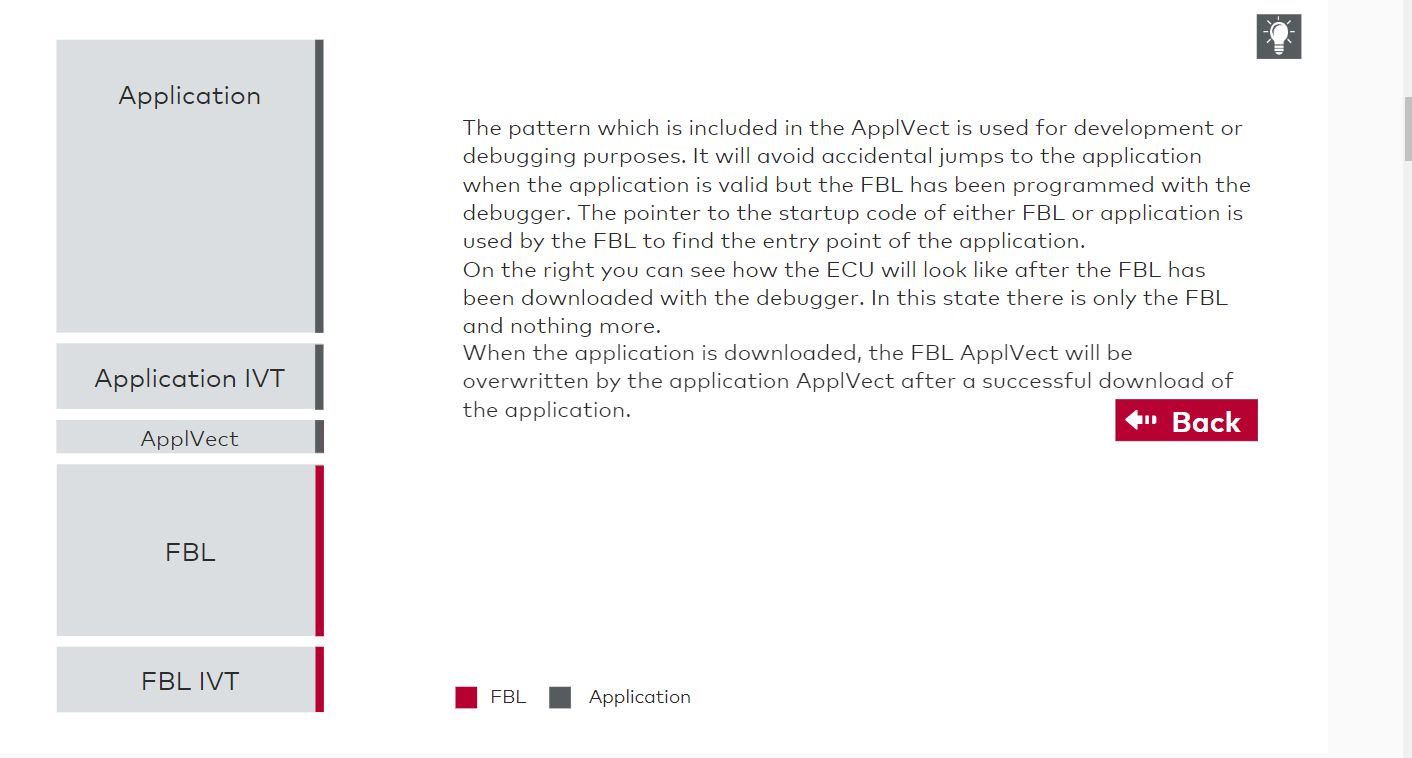
When there is no BM placed in the ECU, the FBL is started as first instance after power-on and software resets. At a certain point of the FBL’s workflow the FBL decides to start the application. To find out the starting point of the application, the so called **Application Vector Table** or short **ApplVect**comes into account.



The ApplVect is an 8-byte array (for 32-bit platforms). It is placed in the application’s memory area and contains:

* A pattern which shows whether the FBL can jump to the application or not (4 byte)
* A pointer to the startup code of either the FBL or the application (4 byte)





**Note** :Even if the ApplVect is available, the application must be valid. Otherwise it is not possible to start the application.

**Attention**

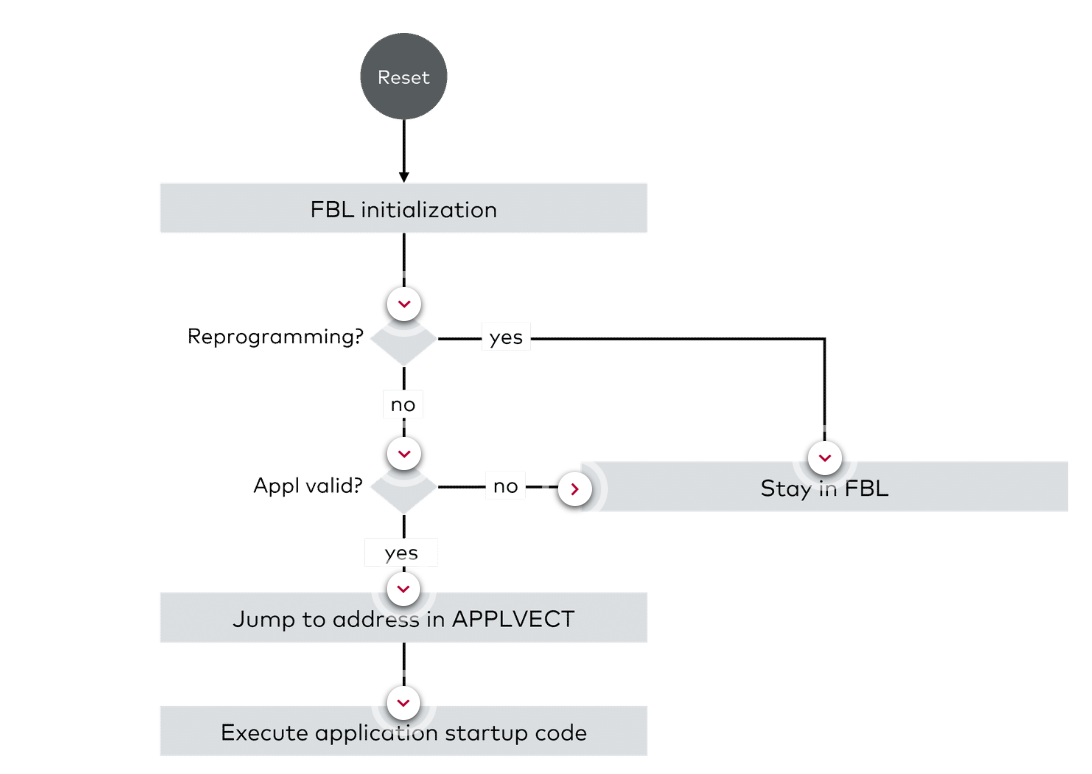
There might be some issues if the ApplVect will

* not point to the application's entry point
* is completely empty
* is corrupt

In such cases, the FBL will still jump to the address located in ApplVect . Erroneous linking of the application may cause exceptions or execution of improper instructions.

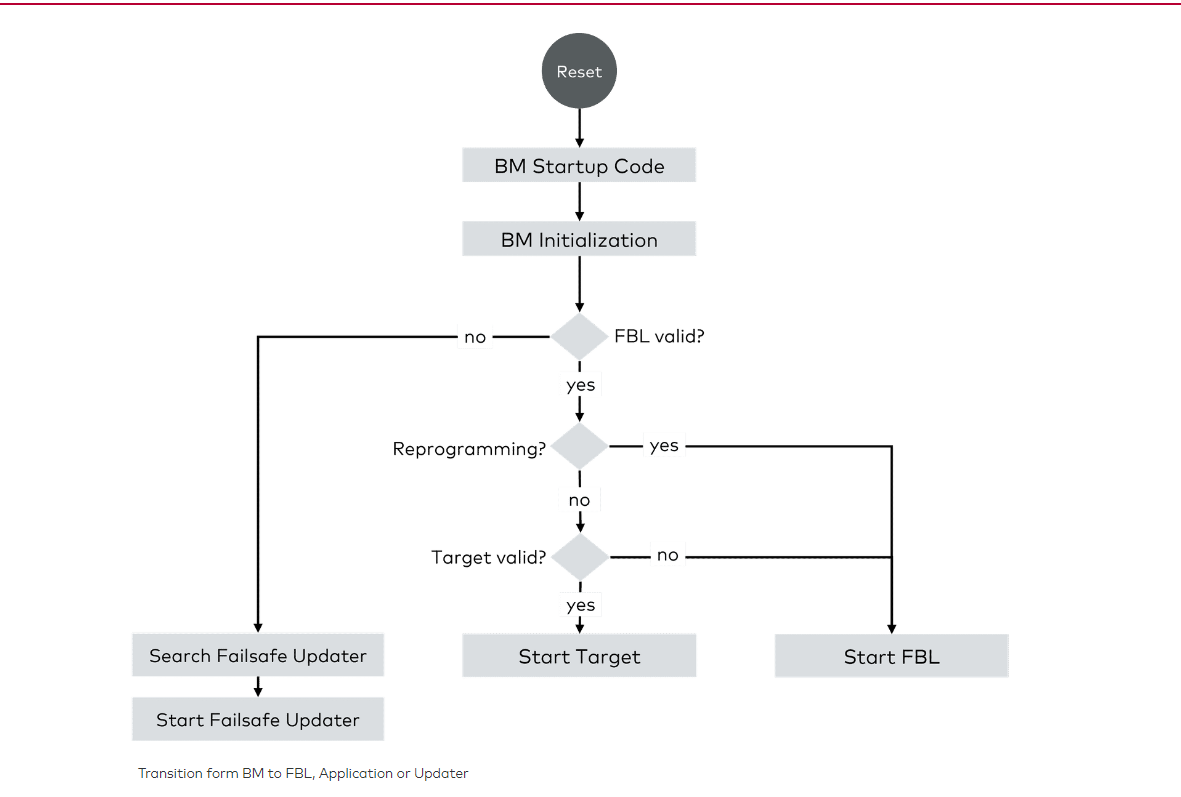
**From FBL to Application (without BM)**

After a power on or any kind of reset, first the FBL is initialized. During the FBL initialization steps, the FBL checks after each startup whether a reprogramming request is available and whether the application validity is set or not. The complete sequence is shown below.



**From BM to FBL, Application or Updater**

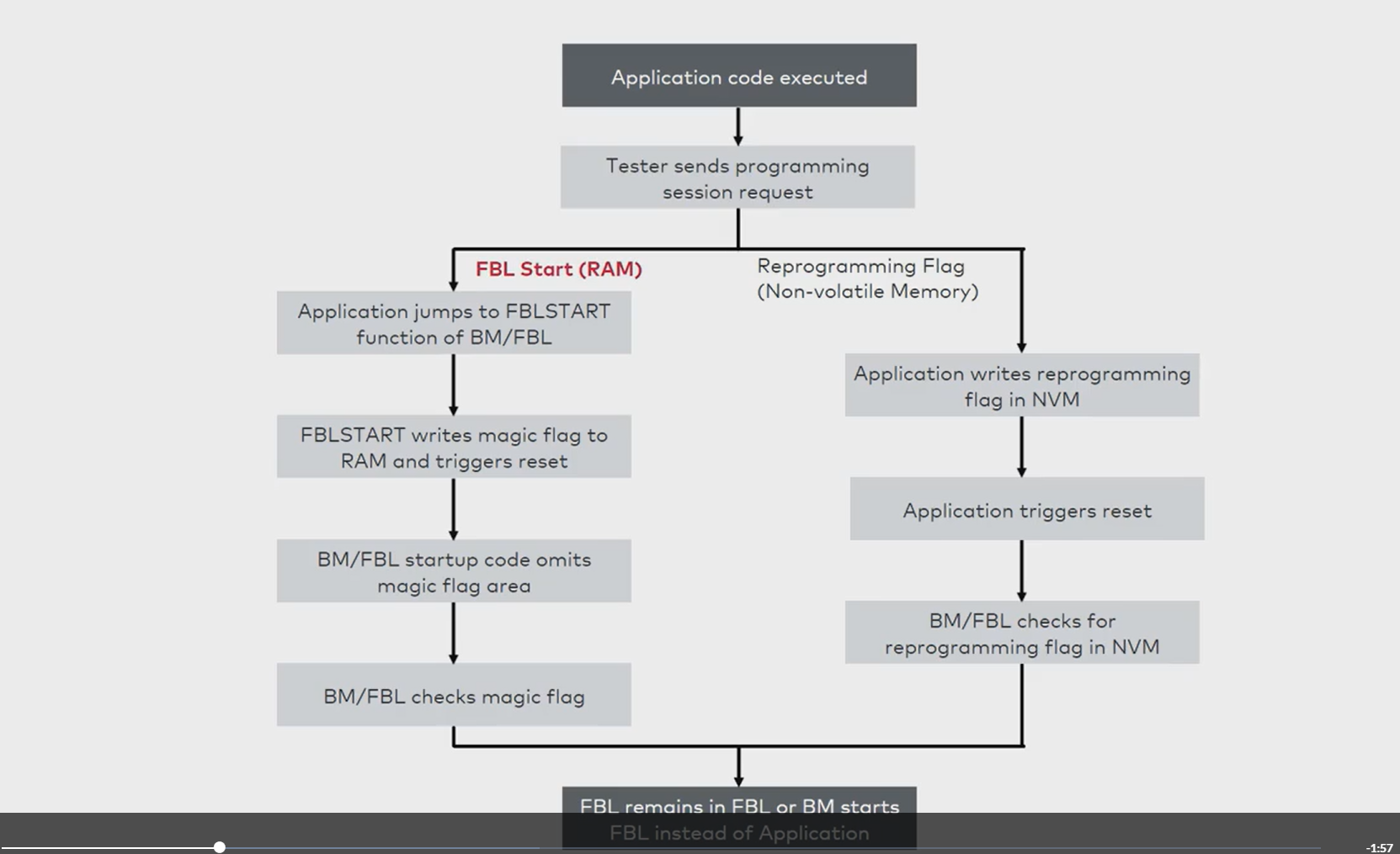
Now let's have a look how the sequence is changed when a BM is part of the ECU. The following graphic will give you a first impression, but the differences are also described down below.



If a BM is part of the ECU, the BM’s startup code is executed quite directly after resets. When the BM is initialized, it will check whether the FBL is valid or not. If no valid FBL is available on the ECU, it will search for a Failsafe Updater.  
With valid FBL the BM checks for a reprogramming flag. If the flag is set, the FBL is started. If the flag is not set, the BM will search for a valid startable target. In case a valid target is found, the BM will start it. Otherwise the FBL will be started to make sure the ECU is able to be (re)programmed.

**From Application to FBL**

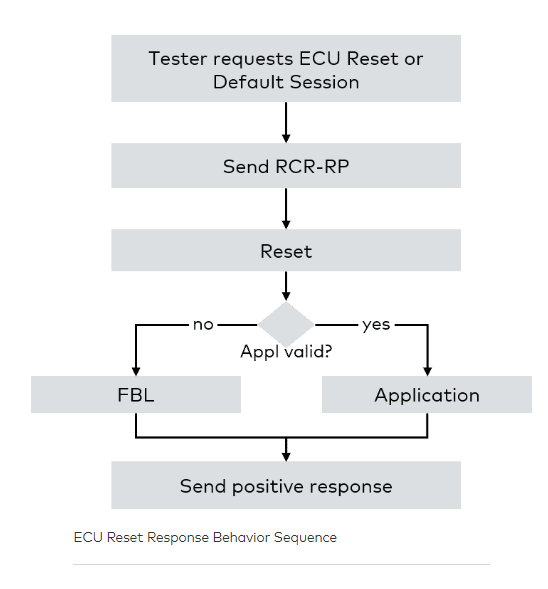
For reprogramming purposes, the application must be able to jump back to the FBL. There are two different approaches to share the reprogramming information between application and FBL, either the feature FBL Start or the use of a reprogramming flag in non-volatile memory. The following video will explain the two approaches.



# Response after Reset and Data Sharing

**Response after Reset**

Another feature of the FBL is the **"Response after Reset"** feature.  
Let's assume the tester sends an ECU reset or default session request. In this case the application should not send a positive response immediately. If it would do so, the tester would directly go on with the download procedure.  
This can cause issues, because the application will trigger a reset. The necessary time of the reset procedure highly depends on characteristics like controller architecture (platform/derivative), startup code execution time (RAM initialization) or BM/FBL initialization time (e.g. FEE, communication stack, secure boot, etc.).



Depending on the P2 and P2\* timing parameters the tester requests must be responded within P2 timing (e.g. 25ms). Most often the duration for the reset procedure takes more than P2 which requires the ECU to send a response pending for requests which issues resets like ECU Reset ($11 $01) or Default Session Request ($10 $01). Therefore, the FBL and application must send a Request-Correctly-Received-Response-Pending (RCR-RP) before they trigger a reset. By sending an RCR-RP the ECU gets more time to perform a reset. Responses of RCR-RPs must be sent within P2\* timing (e.g. 5s).  
After the reset the started program will send the positive response.

**Note:**To find out whether to send a positive response for ECU reset or default session request, shared memory is used similar to the reprogramming flag.

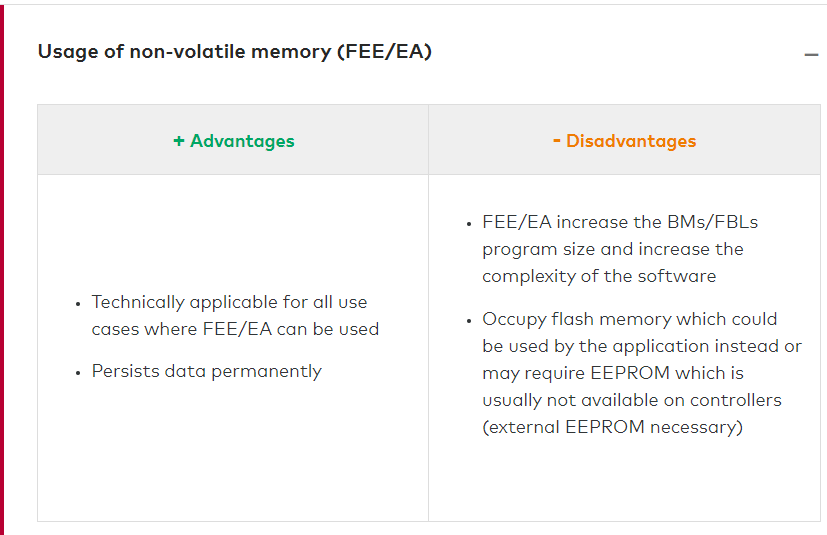
**Attention:**Not all OEMs use this feature. When this feature is not used the application sends the positive response immediately and the tester must wait a specific time until the download procedure continuous. During this period, the ECU resets which may take some time to be fully operable again. This also applies for programming session requests in the application where the ECU performs a reset.

**Data Sharing between BM, FBL and Application**

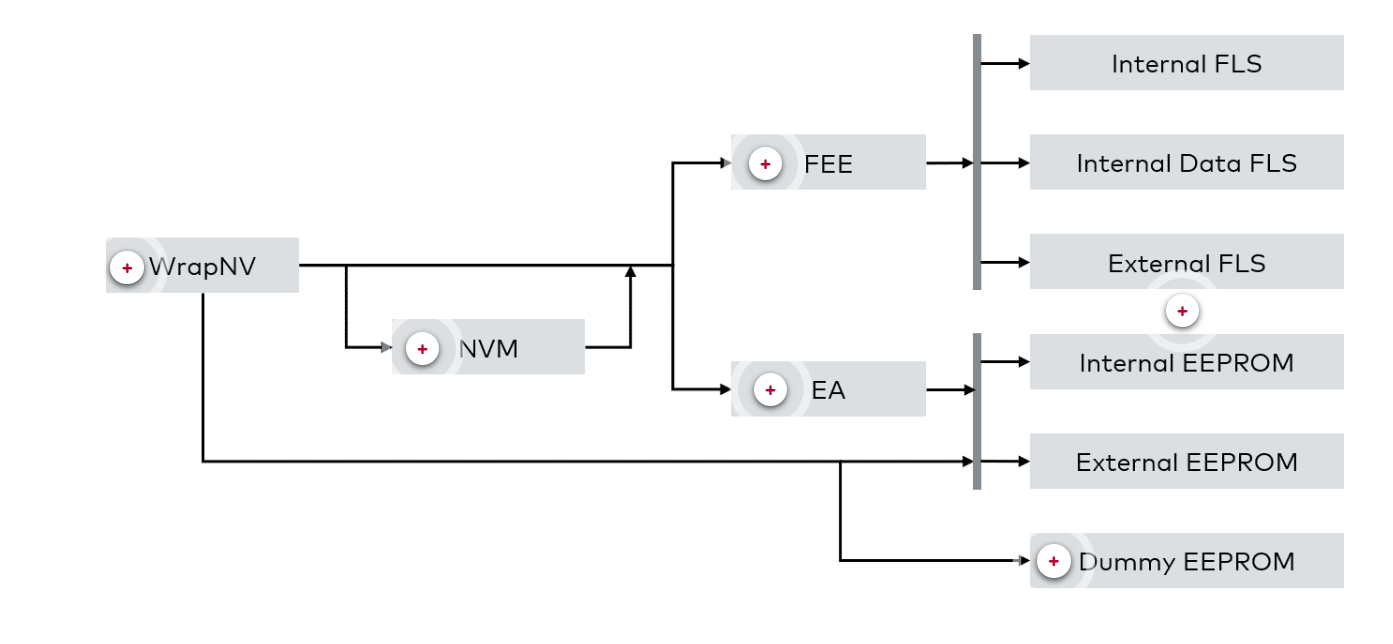
To enable the implementation of features like FBL Start or Response After Reset, data sharing between the BM, the FBL and the application is necessary. Other than that, more data can be shared.

First, let's look at the advantages and disadvantages of the two concepts:





In the graphic below you get an impression how data sharing works.



# Stay in Boot

**Stay in Boot**

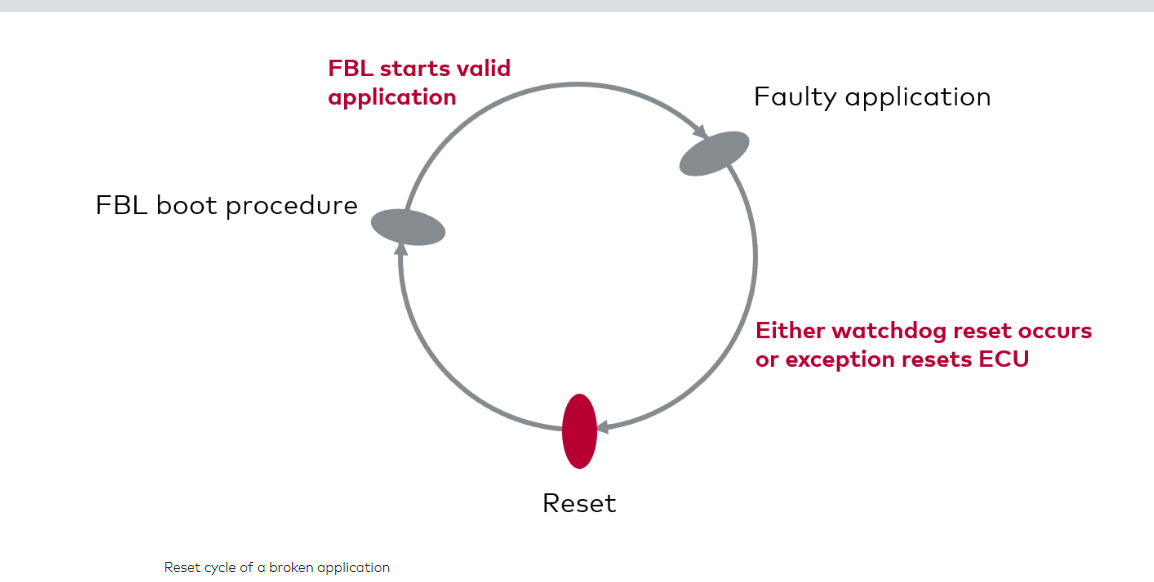
Now we want to discuss another feature called **"Forced Bootloader Startup"** or also known as **"Stay in Boot"**. This feature allows the tester to reprogram software even if the application is broken. Usually this feature is used for development purposes and a few OEMs require this features as well.

**Question:**What are some scenarios where the Stay in Boot feature is usefull? Try to think of some scenarios that lead to a brocken application and then check the examples we compiled down below.

**Some scenarios of a broken application**

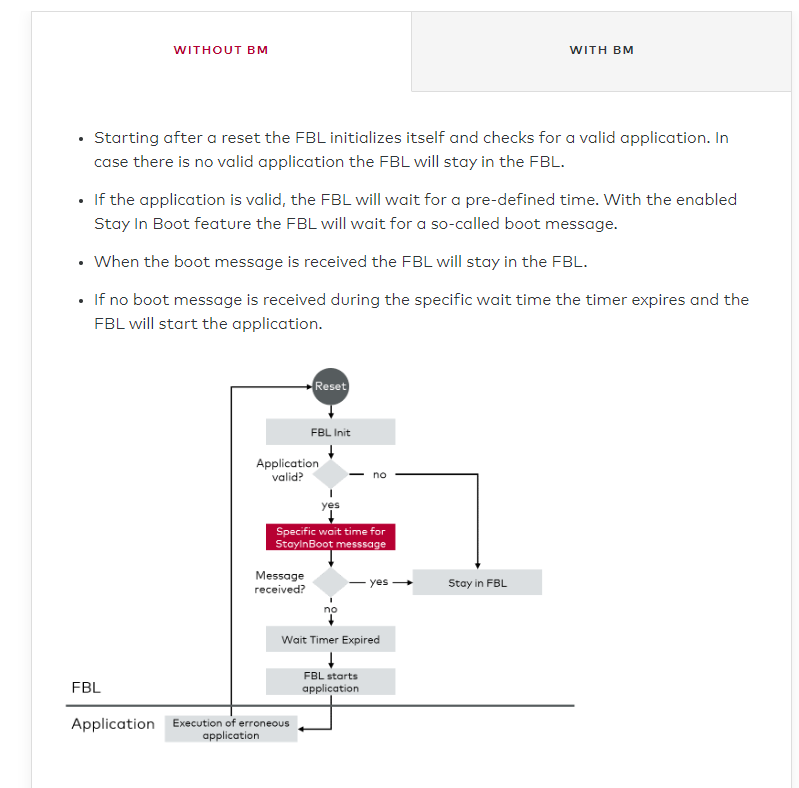
–

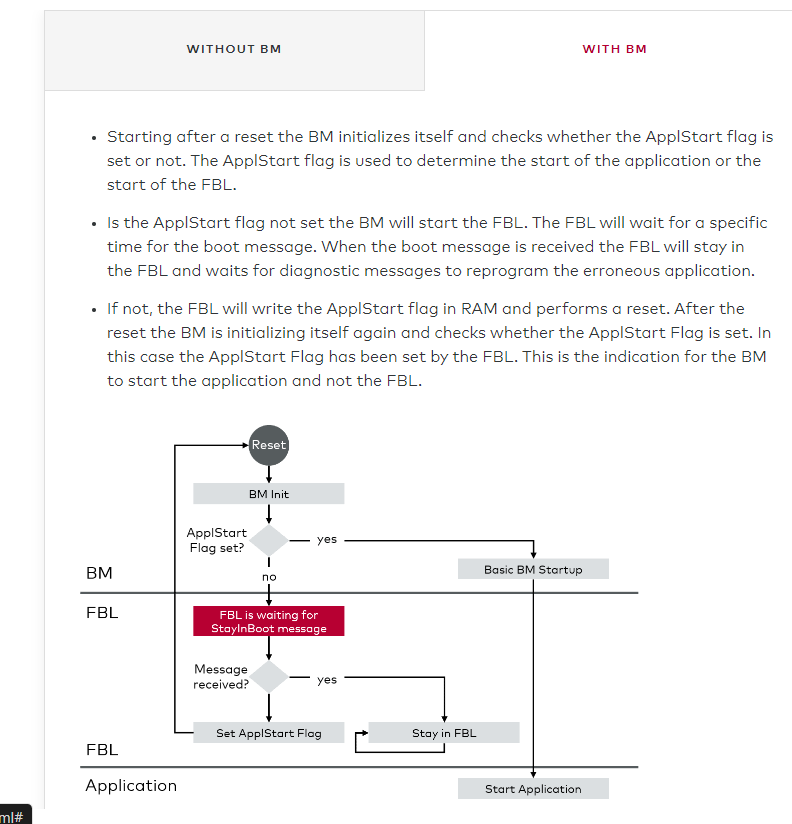
* Communication is not working, e.g. communication stack is configured wrongly, transceiver is not taken into operation
* Application causes an exception before it can descent to the FBL
* Application’s watchdog triggering is faulty
* Descent to FBL is erroneous



The result of all the given scenarios is an **endless reset loop** in which only a connected debugger or the Stay In Boot feature could reprogram the erroneous software.

To mitigate the impact of a broken application, Stay In Boot **prolongs the startup phase** by waiting for several milliseconds (e.g. 40ms) for a **special boot message** from the tester.





# Flash Driver

**Flash Driver**

The Vector Flash Bootloader can only interact with flash drivers which use HIS standardized APIs. Beside the usage of flash drivers it is also possible to access other memories like EEPROM, external NOR flashes or even NAND flashes if they support the HIS API.  
Below you can find some main features and facts of the flash driver.

**Flash drivers do not include diagnostic service functions.**

–

Depending on the memory device and its feature set the following operations are implemented:

* Erase
* Write
* Read
* Derivative specific functions, e.g. setResetVector, verify, unlock flash programming, write security bytes

Often the read function is not part of the flash driver since the controller can access it directly. Only for ECU safe read operations or external memory devices there is a special read function which ensures that data can be read in a reliable manner.

**The support of small micros with small RAM memory is possible.**

–

Even for small derivatives with little resources a flash driver can be provided and executed from RAM.

**The Flash Driver handling is OEM dependent.**

–

The operational area of flash drivers highly depend on the derivative and the OEM specification.

**No accidental start possible if the application is in operating mode.**

–

If the flash driver is not statically linked and compiled with the FBL it is not possible to accidentally execute the flash driver by erroneous implementations e.g. due to a corrupted stack pointer. When the flash driver is stored as c-array in the FBL area the application cannot execute the flash driver since it is stored XOR-encrypted in the c-array. During the initialization phase of the flash driver, the flash driver’s code is decrypted and copied into a RAM buffer ready for execution. If the flash driver is downloaded by the tester the application also cannot execute the flash driver accidentally since the flash driver is not downloaded by the tester and therefore not available in RAM when the application is running.

**Each Flash Driver has a Flash Wrapper (FLIO).**

–

Each flash driver has an own wrapper component called FLIO (Flash IO). It handles the access to the flash driver itself. The FLIO implements the HIS API and enables the access to the flash driver located in RAM.

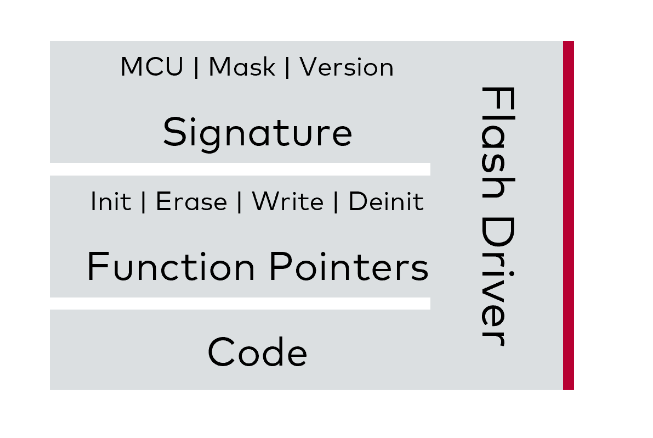
**The BM does not use the Flash Driver.**

–

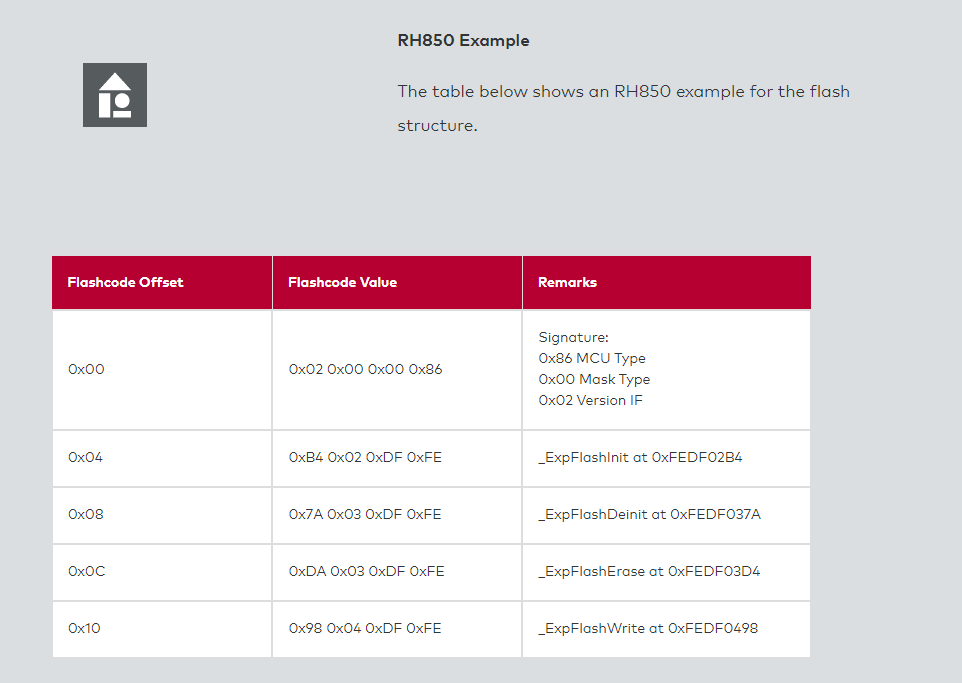
The BM does not use internal program flash driver because it only reads data from flash but does not modify data. There are only a few use cases where the BM operates on memories, e.g. when the Flags are stored in FEE. Usually, the FEE is configured to use the internal data flash and not the program flash to avoid access issues during operations from program flash.

**Flash Structure**

On the right you can see a complete flash driver. Each flash driver has a**flash header**, which includes a signature and the available APIs (function pointers). The **signature** is divided into MCU, the mask and the flash driver version. The API is a **function pointer** structure and will be accessed through the FLIO component. It contains function pointers to routines like init, erase, write and deinit.  
Afterwards follows the actual flash driver code.

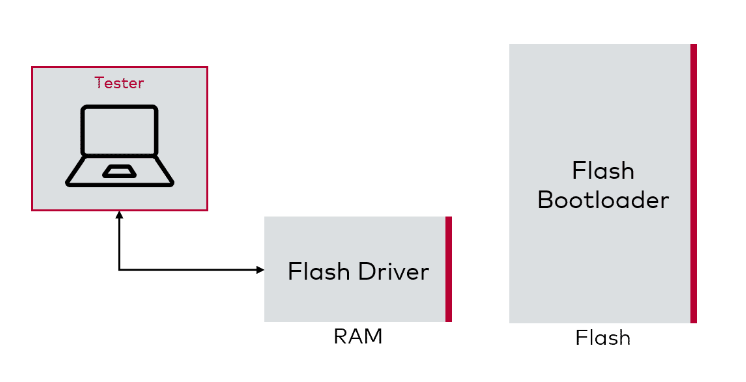


The global RAM buffer **flashcode**contains the complete flash driver code. The starting four bytes are the flash header which contains the signature. Right after the signature the API of the flash driver follows which allows the  FLIO to invoke the respective functions.  The flash header must be linked to the start address of the flashcode. The Bss (block starting symbol) section should be linked after the function pointer section to mitigate impacts of Bss section overwrites.



**Download of the Flash Driver**

When the flash driver is downloaded by the tester the flash driver’s machine code is directly copied to a designated RAM buffer. This RAM buffer is called **flashcode** and is accessed by the FBL during runtime. Right after the flash driver is downloaded, it is initialized to make sure that the FBL can process further requests. This need to be done since the request will require an operational flash driver.



The flash driver is downloaded like a common flash container for the application or calibration data. Depending on the project requirements the container’s data integrity and authenticity is ensured by a checksum and a signature. It may also be encrypted or compressed.

**Advantages**

–

* It is possible to download newer versions of the flash driver in case there are known issues
* Since the flash driver is not stored in program flash with the FBL it won’t occupy flash resources

**Disadvantage**

–

* The OEM must handle an additional container outside the ECU. This may cause additional effort in order to create and manage the flash driver container.

**FlashRom**

When the flash driver is not downloaded with the tester the flash driver is stored in a c-array and compiled with the FBL. The flash driver’s machine code is XOR-encrypted to avoid accidental executions.  
The FlashRom contains a c-array and meta data like:

* The decryption information
* The flash block address and length
* A checksum

**Advantages**

–

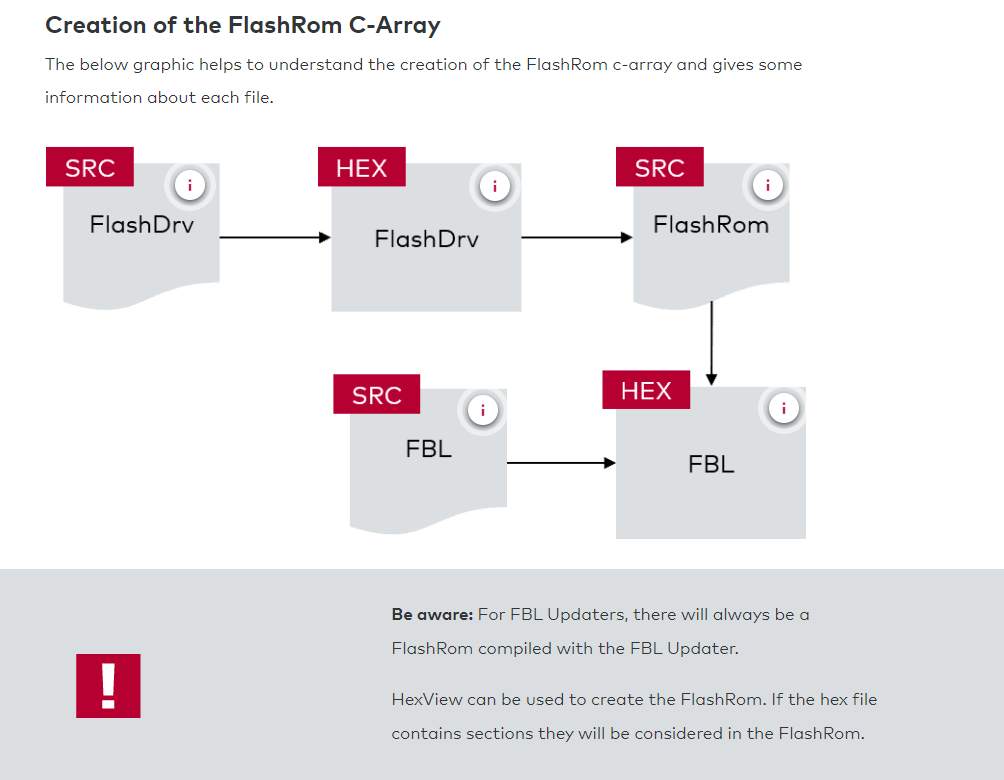
* Flash driver download routine is not necessary because the FlashRom is already compiled with the FBL and initialized before the Flash Driver is used
* Flash driver container handling (creation, signing, etc.) not necessary. Some OEMs avoid handling more containers than necessary because the container-handling creates overheads

**Disadvantages**

–

* When the flash driver has a critical issues the complete FBL must be updated in order to update the FlashRom
* The c-array needs flash memory resource which will increase the size of the FBL

**Note:**Data flash drivers are usually compiled directly with the FBL and are not stored in a c-array. These drivers can be accessed by the controller even when the drivers are executed from flash. In contrast program flash drivers must be executed from RAM to avoid exceptions by the controller.



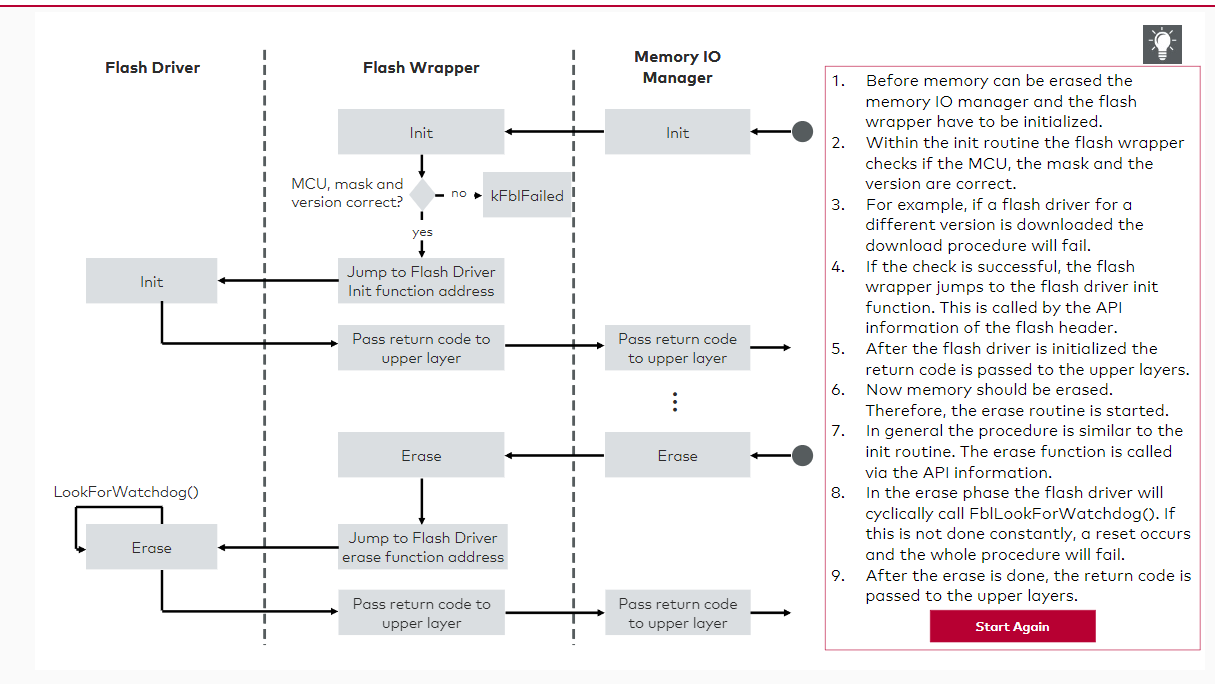
**Flash Wrapper**

Each flash driver has its own **flash wrapper** also called **FLIO**. The FLIO will decrypt the FlashRom c-array during the initialization phase of the flash driver and copy the decrypted machine code to the RAM buffer flashcode.



**Flash Driver Usage during Runtime**

The flash header must be linked to the first address of the flash driver. The version of the flash driver is checked in the initialization phase of the FLIO. When the signature is correct the flash driver’s init function will be invoked. Once returned the available APIs can be invoked as well. The whole procedure is explained in detail in the interactive graphic below.



**Note:**The flash operation of flash banks may differ from the above-described scenario e.g. when the software is executed on flash bank #1 it is possible to perform flash operations on flash bank #2.

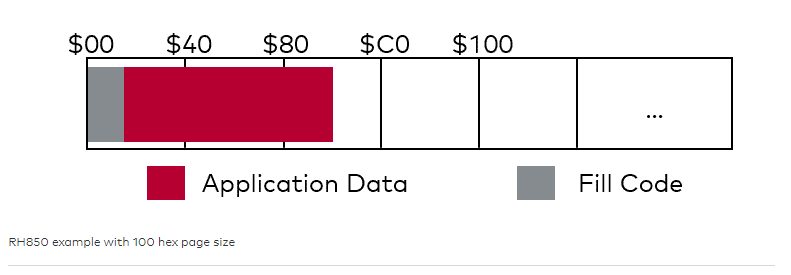
# Flash Segmentation

**Flash Segmenation**

A **page size** is the minimum number of bytes that must be programmed at once. Even if just one byte shall be written to flash the complete page size will be programmed. It is not always possible to align the page size. Therefore, the FBL must do this job.

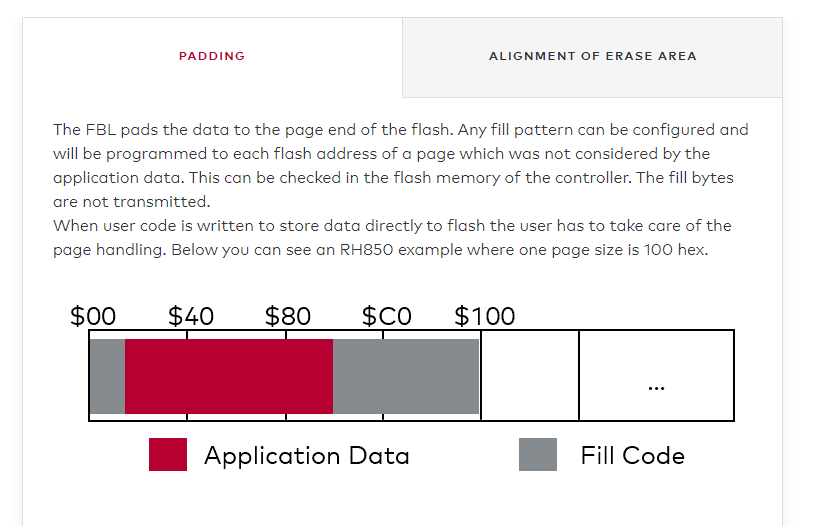
Page sizes differ from controller to controller or rather from flash to flash. Usually, internal program flashes have a quite large page size, varying from about 8 to 256 bytes. Data flash page sizes may start from 2 to 4 bytes and external NOR flash page sizes can even have a page size of 1 byte.

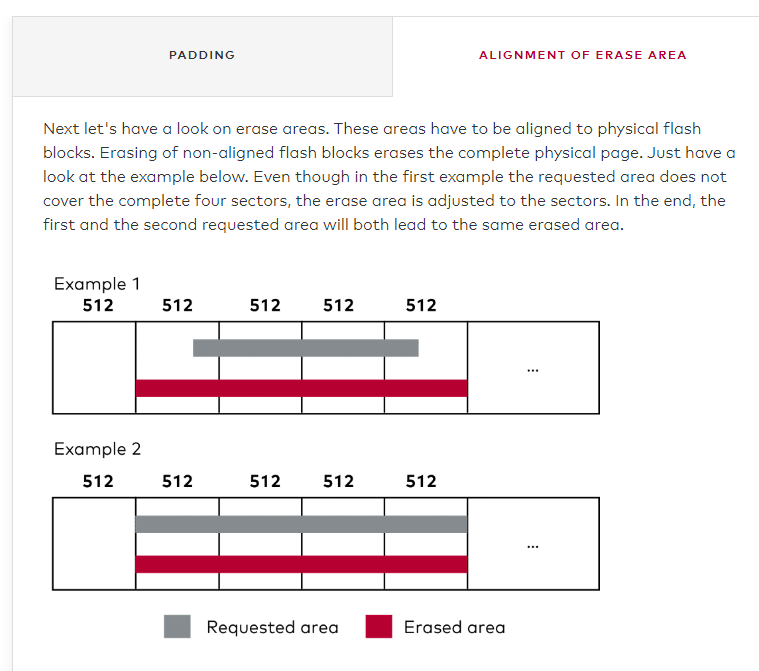
The smaller the page size the better the memory is used in terms of FEE, download container segmentation, presence patterns or any other small chunks of data which will be programmed to flash.

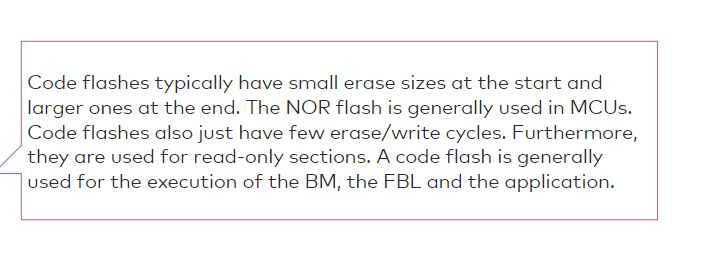


**Note:**Write size/page size and erase size usually differ! Whereas the page size defines the smallest chunk of data to be programmed to the memory, the erase size defines the minimum size of memory to be erased. Additionally, a flash device may have different erase sizes depending on the flash block sizes.

When discussing flash segmentation, there are two aspects that should not be forgotten: **Padding** and **alignment of erase areas**. Both aspects are explained below.









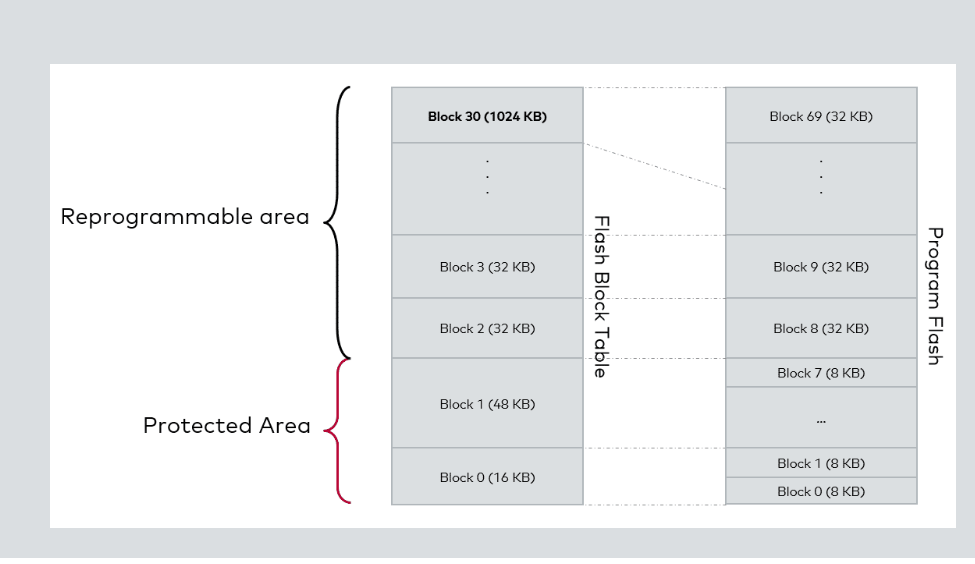
**Flash Block Table**

Now let's have a closer look at the **flash block table (FBT)**. For some derivatives, there are a lot of physical flash blocks available in the memory of the device. To get a better overview, sometimes the upper blocks are summarized in the configured FBT.  
The FBT structure consists of:

* Block start address
* Block end address
* Memory device: The memory device reflects the memory driver to be used for memory operations on this flash block.

**RH850 Example**

Below we have again the RH850 example which can have up to 70 physical code flash blocks (512-1024 physical data flash blocks). The data flash is split up in 64byte blocks but summarized in the config tool to 16KB blocks.



The FBT is used for software-based protection of the FBL. Depending on the configuration/OEM, some parts of the flash must be protected to avoid an unintended overwrite. Nevertheless, the BM for example must know the location of the FBL and must have access to it to read out the validation flags like presence pattern. Without an assigned flash driver, the BM will not be able to check these sections (even if no flash driver is used in the BM). Erase and write address information are verified against entries. If no corresponding area is found, there will be an error message.  
Otherwise also “validation flags” can be used instead of the presence pattern.

**Flash block protection:**

–

* Blocks which are not assigned to a memory device are protected.
* The protection is used for FBL area or OEM/Tier1-specific protected areas.
* The protected blocks are not generated nor can they be reprogrammed.
* The term "protected flash blocks" means that they are in the configuration tool, but they have no memory driver assigned. Therefore, the FBT will not contain these entries and it is not possible to download data to them.

**Entities assigned to protected flash blocks:**

–

* The Boot Manager
* The Hardware Security Module (HSM)
* The FBL (usually if no BM is available)

# Flash Partitioning

**Logical Blocks**

The **logical blocks** are used to summarize flash blocks to one larger block which is used to assign complete data blocks to it (data block = application, calibration, etc.). The normal use case is the erasure of a complete logical block at once. Afterwards the data can be written to this section.  
It is possible to split one data block to several, non-consecutive flash blocks. Gaps in a LB cannot be occupied by another LB. Intersections are disallowed.  
For each block, an own erase and download procedure must be made.

**Attention:**The logical blocks differ from OEM to OEM.

What is a logical block?

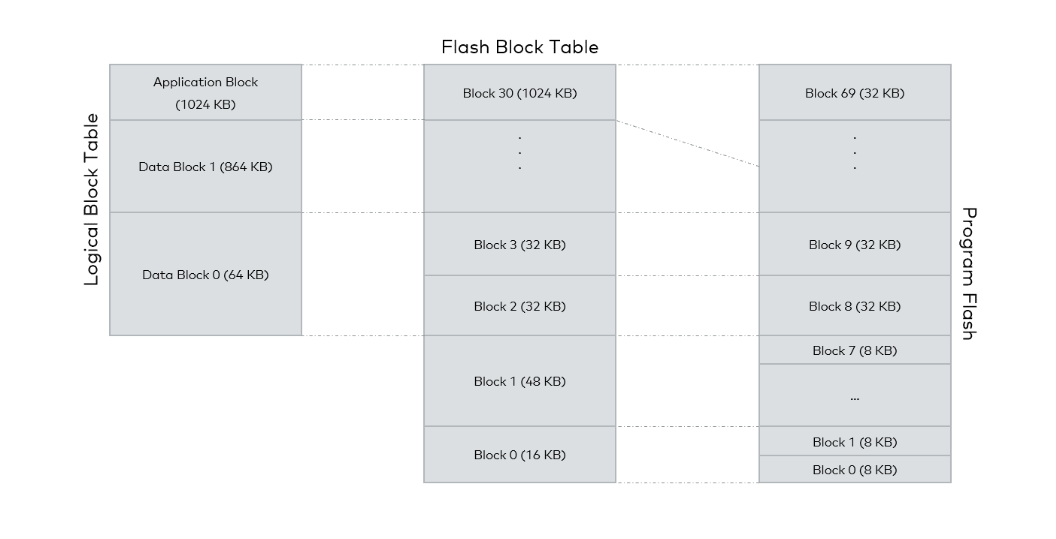
* Physical memory is split in several segments, called logical blocks in the logical block table (LBT)
  + Smallest logical erase section compliant to UDS
  + Logical block corresponds to a hard disk partition for example
* The logical split enables to update the application area in smaller chunks e.g. if only the calibration block is affected the application software does not need an update

How are logical blocks defined?

There is a descriptor table, called logical table (LBT) which corresponds to the partition table of a hard disk.

What is the intention of the logical block?

* Application area can be split into parts
* Parts which are changed frequently e.g. parameter sets
* Stable parts like the application itself
* **Flash Block Table and Logical Block Table**
* Let's have a look at the connection between the **flash block table (FBT)** and the **logical block table (LBT)**. The FBT can consist of a summary of the real program flash blocks. In the LBT each logical block can contain more than one flash block of the FBT. The LBT is part of the FBL and contains partitioning information.



Logical blocks are a summary of underlaying configured flash blocks which then are mapped to the hardware flash blocks.  
For each flash block in the FBT, there is a memory driver assigned. Meaning even if a logical block contains different flash blocks with different memory drivers, it is possible to download them.  
This will be handled by the MIO (Memory IO Manager). It will check which memory driver is used for which flash block and then sends the data to the responsible memory driver.

**Structure of LBT**

The LBT consists of **header information**and **block descriptors**. The LBT structure consists of:

* Block number
* Block type (mandatory/optional)
* Block start address
* Block length
* Max. programming attempts
* Verification functions
* In case of BM: BmHrdHeader address

**Start address and length**

–

The start address and length will be used for a complete erase. Also, if you program to this logical block, it will check whether the requested data fits into the LB area.

**Block type**

–

The block type will be used to validate the ApplValidity, which means if the application is valid and can be started.

For example, you have an application and a calibration block which need to be in the ECU (mandatorily) because the application depends on the calibration block. In this case you must make both LBs mandatory in the configuration tool. This will lead to the case, that if only one of these blocks are downloaded, the application validity is not set. Therefore, the diagnostic routine "check programming dependencies" will return an NRC (Negative Response Code). But afterwards, if you download only the other mandatory block, everything will be validated the application can be started.

**Verification functions**

–

These functions can be used to verify the logical block. This can be a verification at the end of the download or also a pipelined verification which is started next to the download to speed up the download itself.

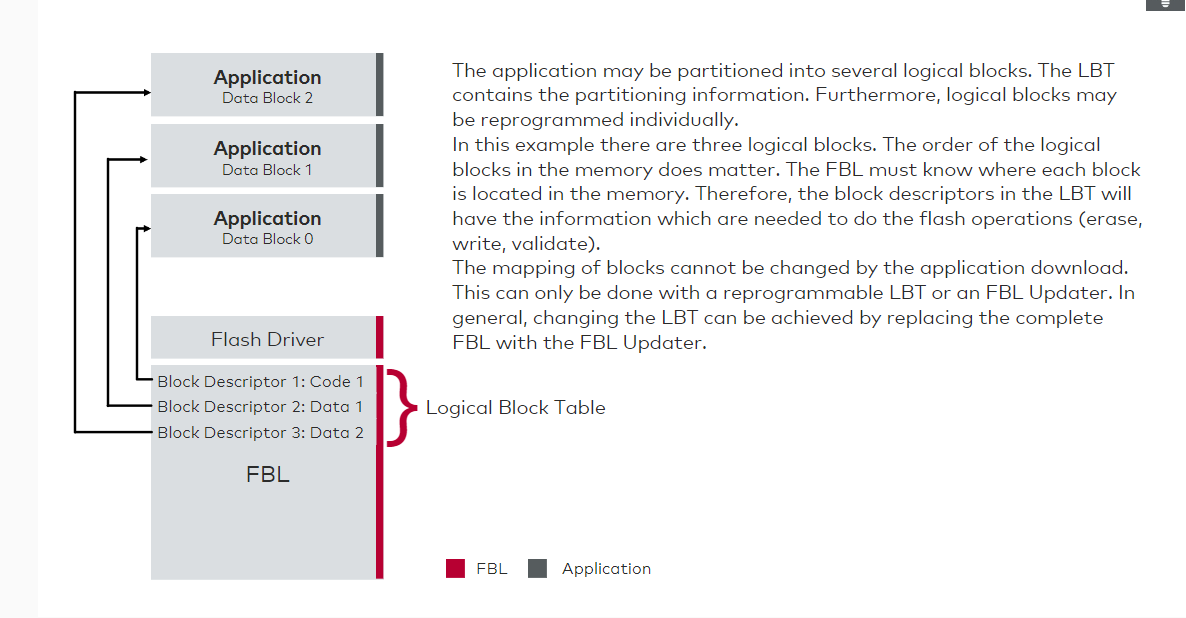
**BmHdrHeader**

–

This address is checked by the BM to find the target's entry point.

**Memory Layout**

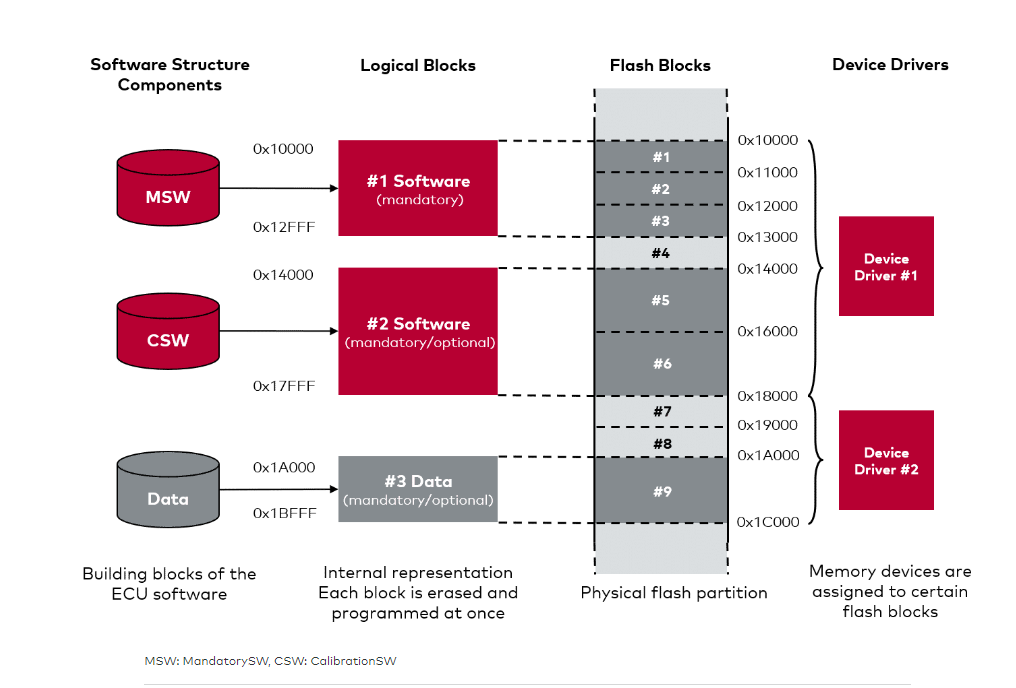
Now let's have a look at the memory layout.



**Memory Partitions**

**Memory Partitions**

The OEM could split the memory into different software components. These software components are linked one-by-one to a logical block. As you should already know, the logical blocks are mapped to several flash blocks. These flash blocks then again are linked to a special device and use a special device driver. To better understand this have look at the graphic below.



An example for the **memory partitioning** you can see above would be an ECU which uses pcitures and audio files. Media files take a lot of data and are usually stored to an external program flash. Therefore, the data would be linked to an external flash driver.

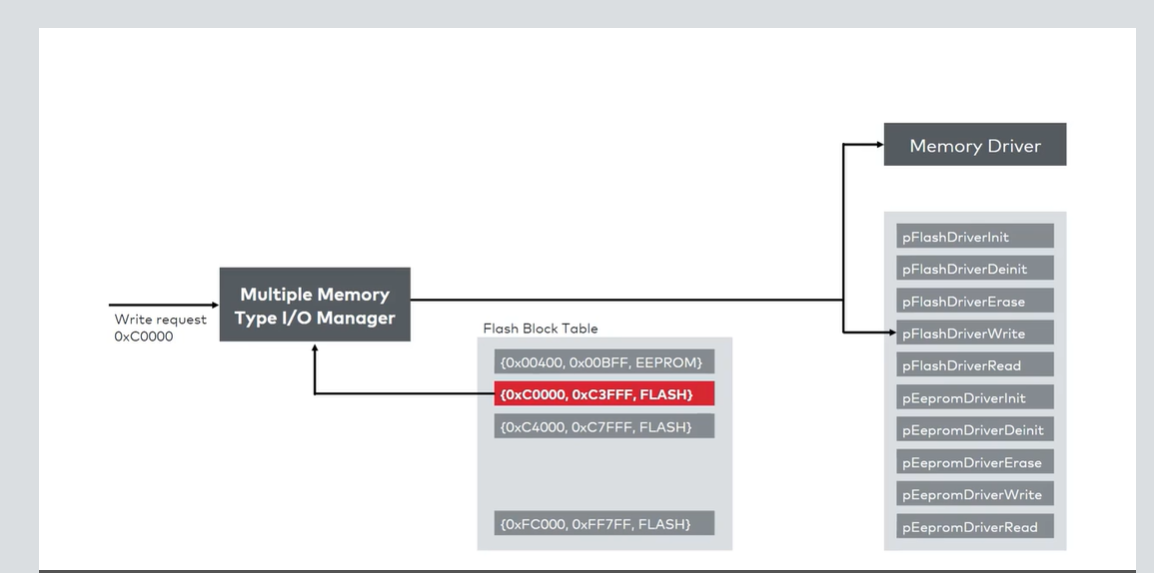
Furthermore, the addresses can have a virtual address. In our example the device 2 is an external flash, which should be programmed at 0x00 at his side, but the LBT points to 0x1\_6000-0x1\_C000. Therefore, a remapping of the virtual address in the LBT must be done to have the address matched in the external memory to 0x00.

Let's assume you have one application which uses a part in the EEP. It can be set up like in the picture. The download will then start with the flash driver to the first segments and later with the EEP driver to the EEP address.

**Multiple Memory Devices**

Let's have a look how these multiple memory devices are handled.  
Memory drivers are selected by using a **memory I/O manager**. The memory manager selects the needed memory driver in case of an erase, write, read cycle. Which means, if a logical block consists of multiple memories, it will use the needed memory driver for the operation on the different memories.

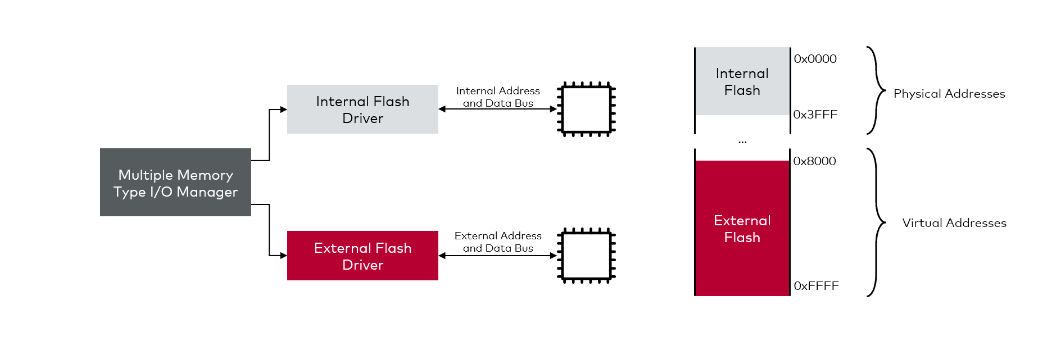
In the FBL this multiple memory manager is called the MIO (Memory IO Manager).



**Hint:**If a driver is not initialized when it should be used, the multiple memory manager will initialize the driver if needed and use it then. Afterwards the driver will be deinitialized or erased e.g. the flash driver is downloadable, this driver will be erased from the RAM after a complete download process.

**Use Case with External Memory**

Let's discuss another example. The below example shows support of internal and external flash. The external flash is a SPI/I²C EEPROM. Keep in mind that the external flash may not have that large addresses like shown in the picture. Therefore, a remapping of the addresses must take place in the driver. Meaning for example the FLIO (Flash IO, this might not be a flash (EEP) as well) of this device will subtract an offset from the requested write address to match the physical address.



The validation of an external attached memory device can take a lot of time. This should be considered because you must read back the written data to calculate e.g. a CRC or a signature. If this happens over SPI, it will take time to transmit everything and to read it back to the host ECU. Here the bottleneck is most of the time the SPI communication. To speed this up, some external memories e.g. a SBC or a microcontroller, can calculate the CRC on their own.

The software components are linked one-by-one to a logical block. The logical blocks are mapped to several flash blocks. These flash blocks then again are linked to a special device and use a special device driver.

The memory manager selects the needed memory driver in case of an erase, write, read cycle. Which means, if a logical block consists of multiple memories, it will use the needed memory driver for the operation on the different memories.

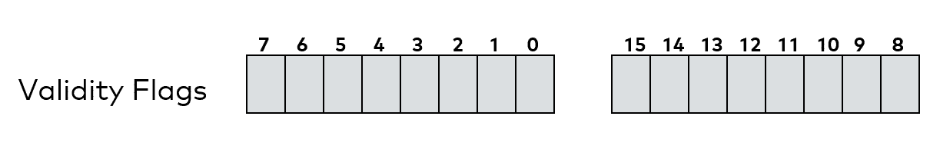
If a driver is not initialized when it should be used, the multiple memory manager will initialize the driver if needed and use it then.

# Block Validation

**Block Validation Flags in NV-memory**

The first approach is to store the block validation information in **non-volatile memory** like EEPROM or flash. For the EEPROM each logical block is represented by one flag, represented in a bit mask. The FBL can determine application validity at once. Meaning there are no loops over logical blocks necessary.

The validity flags for the logical blocks are stored in a byte as **single bits**, therefore only 8 blocks are allowed to be present in the LBT.



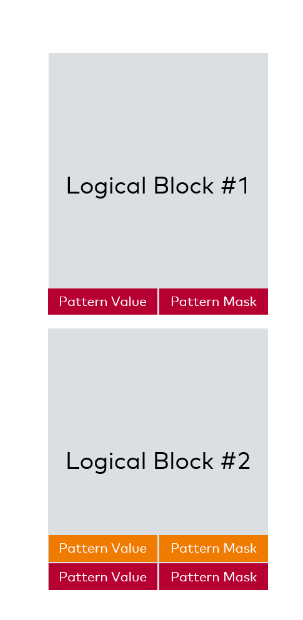
**Presence Pattern**

If the EEPROM shall not be used, a presence pattern (PP) can be used instead. With PP the validity information is **stored with the logical blocks in flash**. This is the second approach to handle block validation.

PP block validation is used to check whether a block or even the complete application is valid. Let's assume the calibration block and the application block are mandatory. If only the application is downloaded, the application block itself is valid, but not the application validity. Later, you can download the calibration because the application block is already valid. When the calibration will be validated, all necessary parts in the ECU have been programmed and the application validity can be set to true.  
**Note:**Often the application validity is set in a dedicated service e.g. CheckProgrammingDependencies.

For this example, there is a second PP block in the second logical block. This orange block is the application validity. This application validity will be written in the last downloaded block if all mandatory blocks are valid. Which means also this pattern can also be in the #1 block if this would be the last downloaded block.

If PPs are enabled, writing to the PP is prohibited during the download. 4\*PP size is reserved at the end of the logical block. The FBL will refuse the reprogramming to this area if requested by the tester. Though, the erase of the logical block is applicable. For example, the flash segment size is 0x20, therefore 0x80 will be reserved at the end of the logical block.



**Difficulty of presence pattern**

–

The flash must be erased before programming. Meaning that the presence pattern will be erased as well. If the flash peripheral starts with the erase of the first flash blocks and an unexpected reset occurs the first flash blocks will be erased but not the last where the presence pattern is still in valid state.

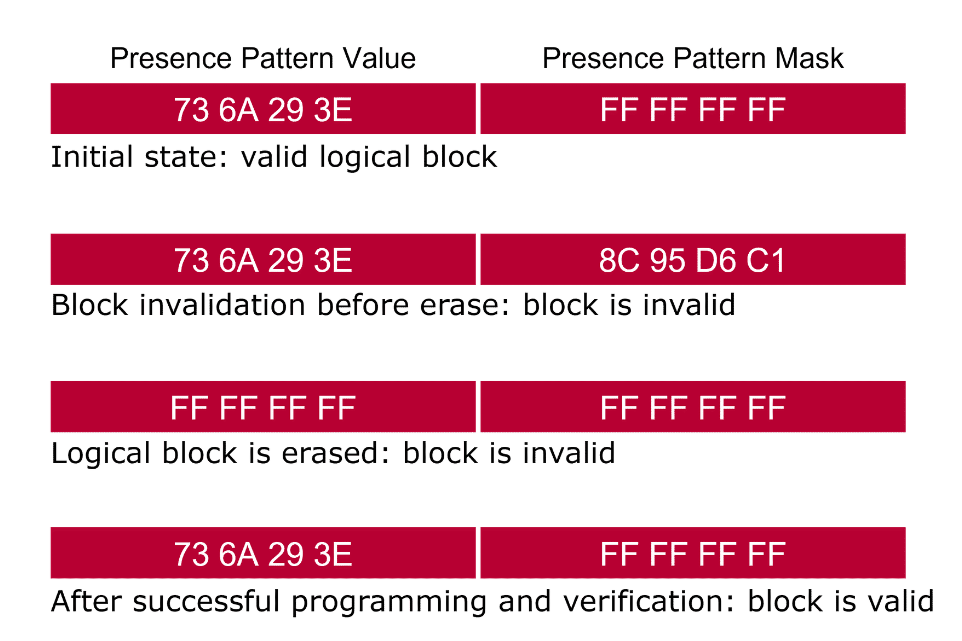
**Solution for the difficulty**

–

There are two patterns at the end of a logical block: mask and value. Each pattern is aligned to the flash segment size. There are at least 4 bytes used for each pattern.

**States of the Presence Pattern**

To discuss how the PP works in detail, let's have a look at the different **states of the PP**. The example in the graphic below depicts the validation and invalidation flow during a typical download procedure. Be aware that the exact procedure may vary between OEMs.



1. Initially the logical block is erased therefore the presence pattern area is also erased.
2. Once the logical block has been downloaded and verified successfully the function ApplFblValidateBlock is invoked in order to write the presence pattern value to the logical block and check the define kFblNvMarkerValue in the implementation.
3. When the tester or rather the Flash Bootloader flashes a logical block again it will invalidate the logical block before the logical block is erased. The purpose of the invalidation is to mitigate issues that may occur during erasure. For example a power drop which keeps the presence pattern valid although half of the logical block was erased before the issue occurred. To invalidate a logical block the function ApplFblInvalidateBlock is invoked and the presence pattern mask is written, check the define kFblNvMaskValue.
4. During the erase of the complete logical block the presence pattern and mask are erased again like in step 1.
5. Like in step 2 the presence pattern value can be written again due to the prior erase of the presence pattern area.

# Security and Compression

**Security**

There are different standard security features which are already implemented. In most cases the delivery is already tailored to the needed use case.

**Authorization**

–

The SeedKey algorithm prevents unauthorized access. The interactive graphic under the subheading "Authorization with SeedKey" gives you a deeper insight into the SeedKey mechanism.

**Integrity**

–

* Ensures that the transmission of data was performed correctly
* All data bits are transferred and programmed correctly into flash
* Checksum algorithm e.g., CRC32

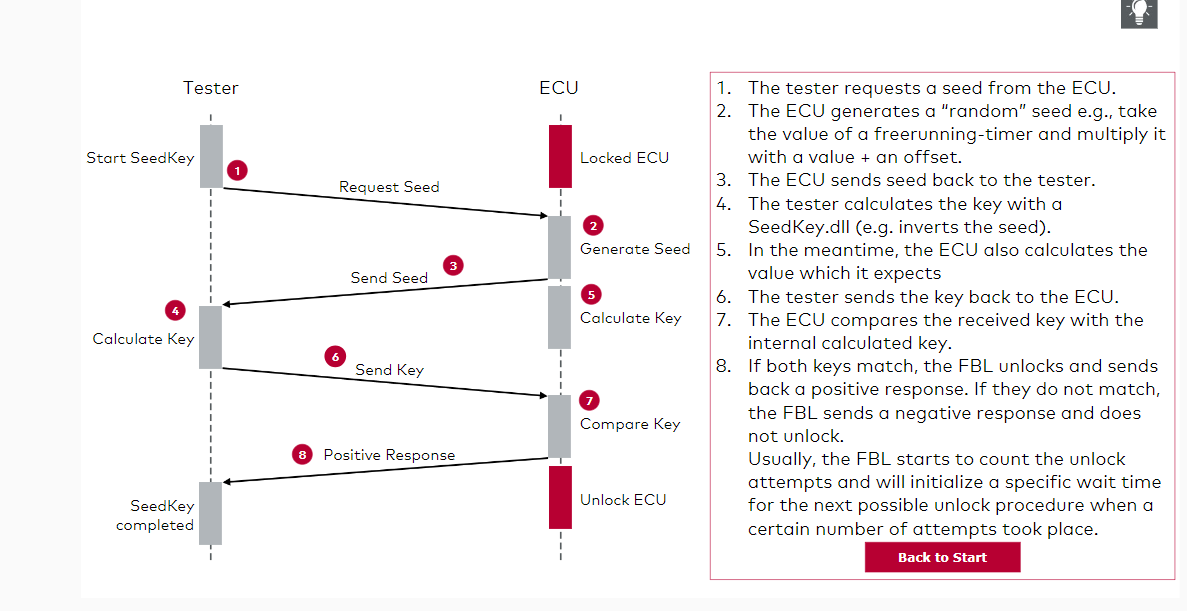
**Authenticity**

–

* Download comes from a legitimate source
* Manipulated software is not accepted
* Signature verification
* HMAC (Hash Message Authentication Code), RSA (Signature)

**Authorization with SeedKey**

The following animation shows the communication between a tester and the ECU for the SeedKey mechanism.

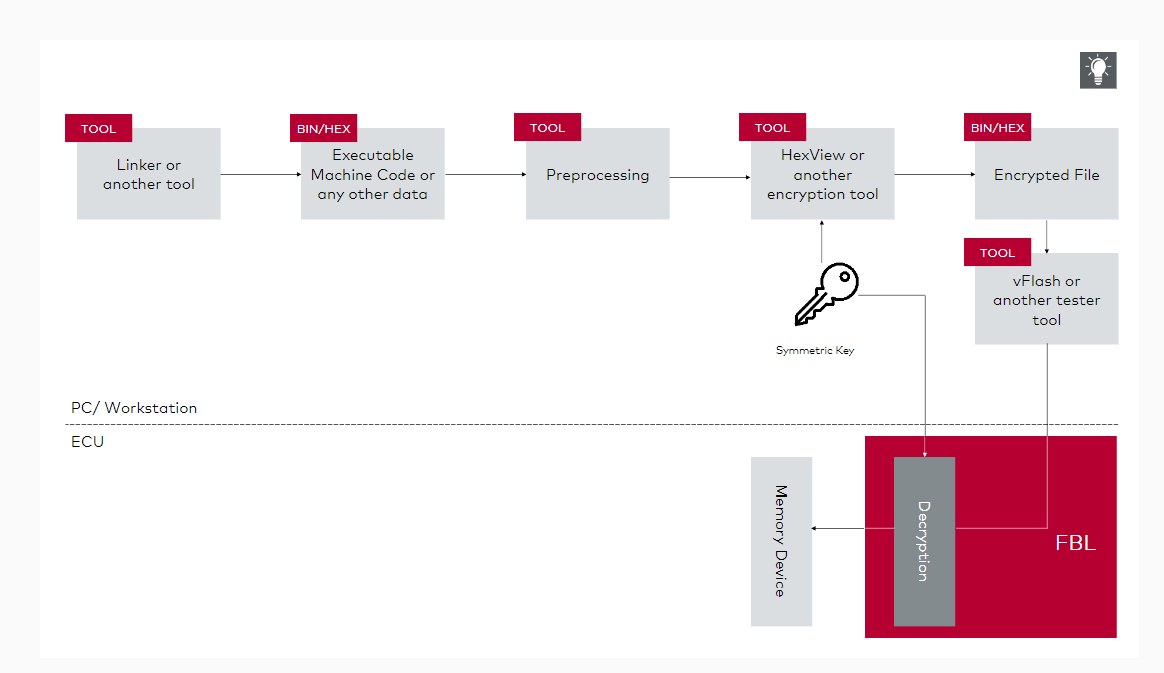


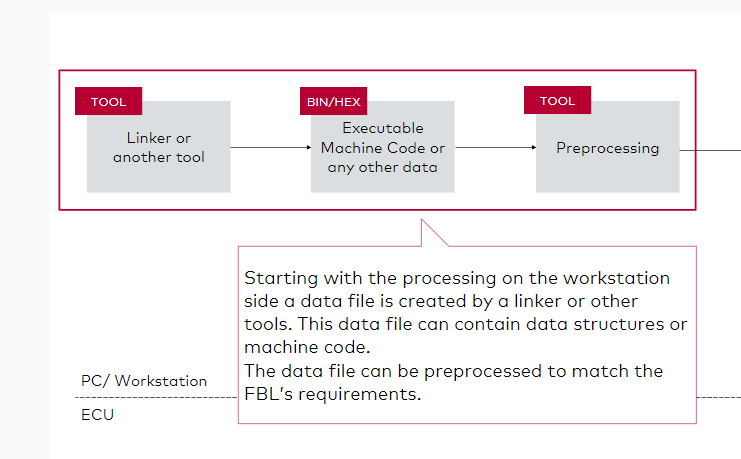
**Procedure of Symmetric Data Encryption/ Decryption**

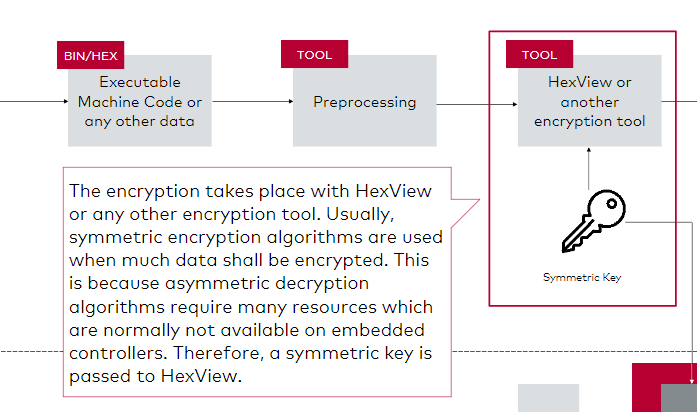
Let's have a look at the procedure of symmetric data encryption and decryption. The following example shows how a symmetrically encrypted download container is created, transferred to the FBL, decrypted and programmed to the memory.

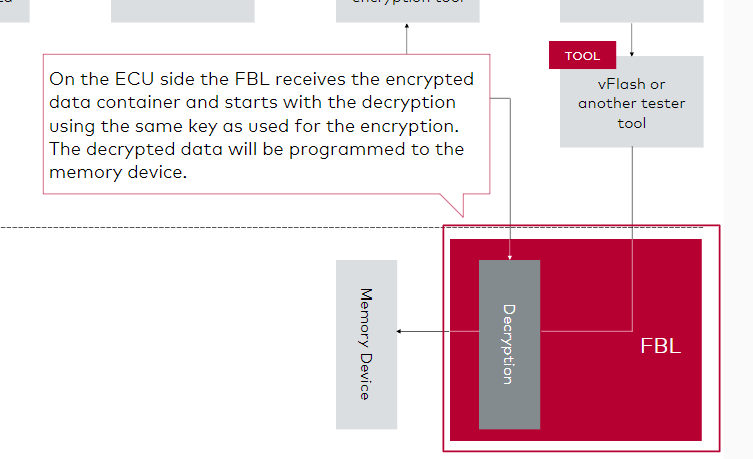
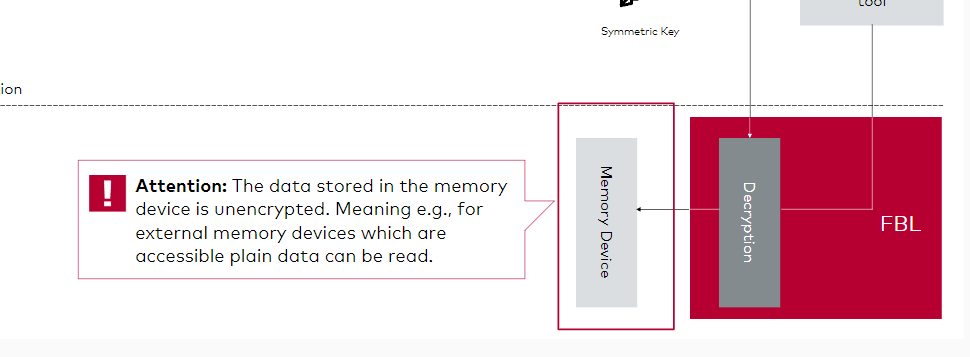
The symmetric encryption only secures the data transmission and not the storage itself. Meaning the plain data is programmed to the memory device.

The symmetric key is used for the data encryption and data decryption. Therefore, the symmetric key must be stored in the FBL as well. The key can be stored plainly in the FBL's flash memory or in a HSM (Hardware Security Module).







**Note**

It is also be possible to program the encrypted data to the memory. The instance which accesses the encrypted data must decrypt the data before it can be used. Nevertheless, this is possible but not standardly implemented.

**Data Compression**

Important aspects that must be considered in case of data compression are

* Compression rate (ratio of uncompressed and compressed length)
* ECU decompression speed
* Memory consumption

For a Flash Bootloader, decompression speed and memory consumption are important parameters for the decompression.  
In general, the compression is mostly used to speed up the download of data over the selected bus system. Unfortunately, not all controllers/ compression algorithms are speeding up the overall flashing time. The reason behind this is that the ECU's time for the decompression of the received data may be more time consuming than the time saved by transferring compressed data.

**Example: LZMA**

LZMA (Lempel-Ziv-Markow-Algorithmus) is a very efficient compression method. Therefore, the data to transmit is often very small, but also the decompression takes a lot of time due to the complex algorithm.

**Caution**

Beware of random values. For example, if a gap is filled with random data, the compressed data will be larger than the original data.

The compression needs an additional buffer, which is configured in the configuration tool.  
The reason for this is that the data is transferred to the diagnostic buffer. From this buffer, the data is decompressed and stored in the data processing buffer. Then the decompressed data is written to the flash memory.  
This means that more memory must be available in the RAM for this use case. The buffer sizes also have an impact on the overall download speed.

**Generally, the larger the buffers, the faster the download. However, this is mainly dependent on the derivative. So if the buffer becomes larger than a certain size, the improvement will not be as strong.**

**Hint**

For some compressions, there is also a configuration file that can be adapted for specific buffers or speedup mechanisms.

The following compression modules and algorithms are available:

* Vector (LZSS)
* LZMA
* Third party compression modules

Third party compression components are not part of the FBL SIP and must be obtained from the OEM.

**LZSS**

Vector uses a special variant of LZSS for (runtime/code size) optimized decompression within the ECU which is optimized for embedded use cases. Typically, a download time saving of about 20-50% is possible.

**Example: Resulting implementation from Vector Informatik: Full decompression on Motorola Star12, @16MHz (no PLL)**

This Vector implementation of LZSS can help to have a low resource consuming compression component which can speed up the whole download process

* LZSS is based on the LZ77 algorithm.
* ~ 600 Bytes of ROM (low memory consumption)
* Compression algorithm for powerful micros also available (900 Bytes ROM for implementation).

This example is mentioning an old and slow derivate and shows that also these ECUs can handle this decompression in a reasonable time.

**LZMA**

LZMA is a free compression algorithm that is used in standard desktop tools like 7Zip.  
As mentioned before, this algorithm has a good compression ratio for repetitive data patterns (small download file size) but needs more time to decompress on ECUs then LZSS. Therefore, there is a configuration file where several options can be changed to improve also the speed of decompression, e.g., a larger RAM buffer or less watchdog calls.

**Caution**

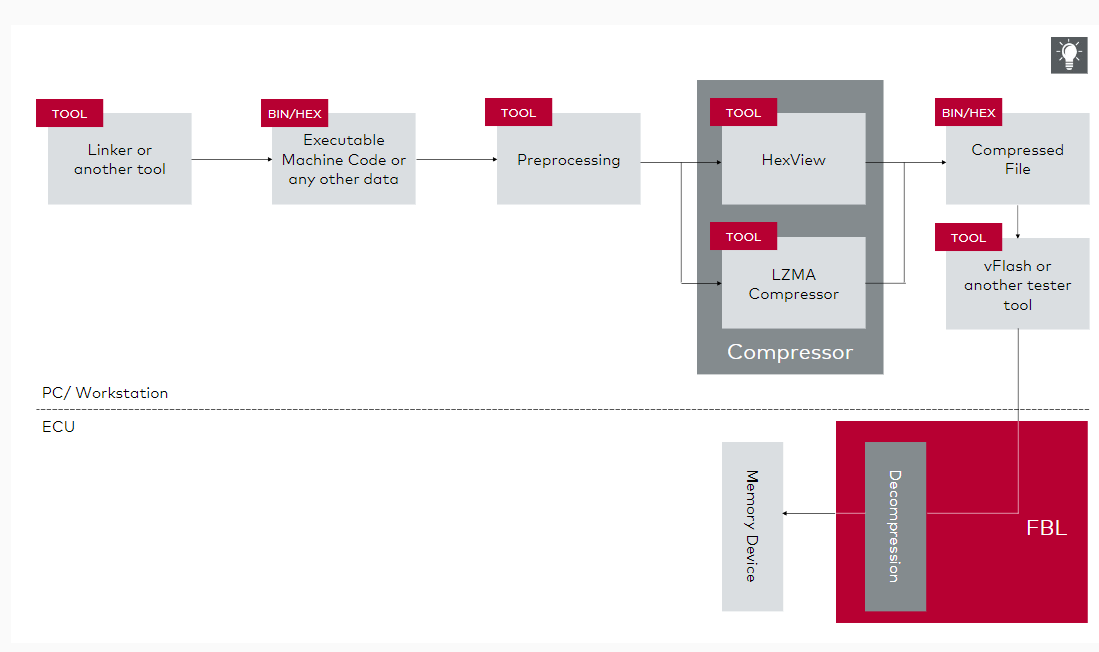
FBL runs in polling mode and therefore the FblLookForWatchdog function must be called cyclically.

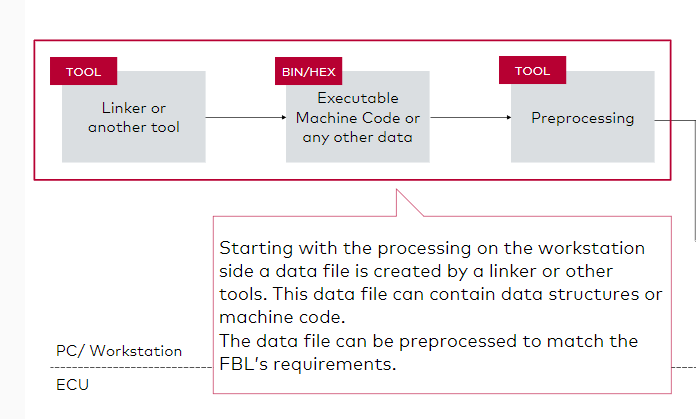
**Caution**

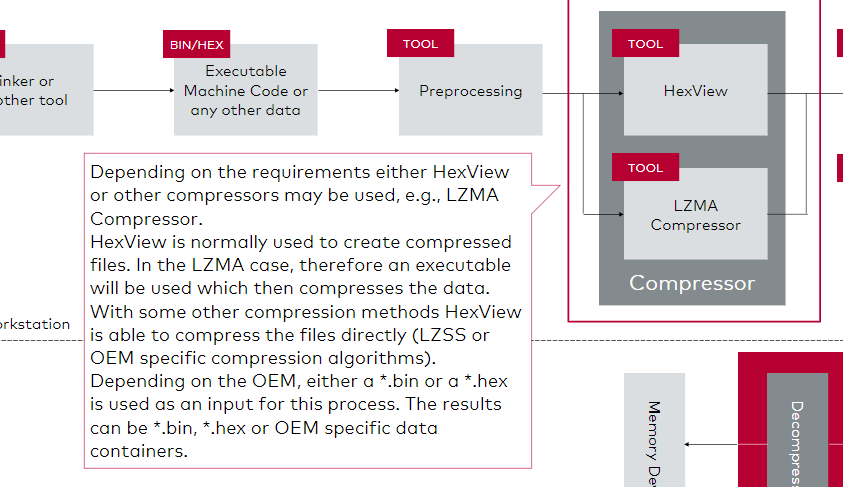
The function FblLookForWatchdog needs to be called in every long-lasting function to ensure correct timings and watchdog triggers. If the call of the watchdog function is set to less often, it may happen that the watchdog is no longer called in time and thus triggers a reset on the ECU or misses some cyclic tasks which are related to the functionality of the Bootloader.

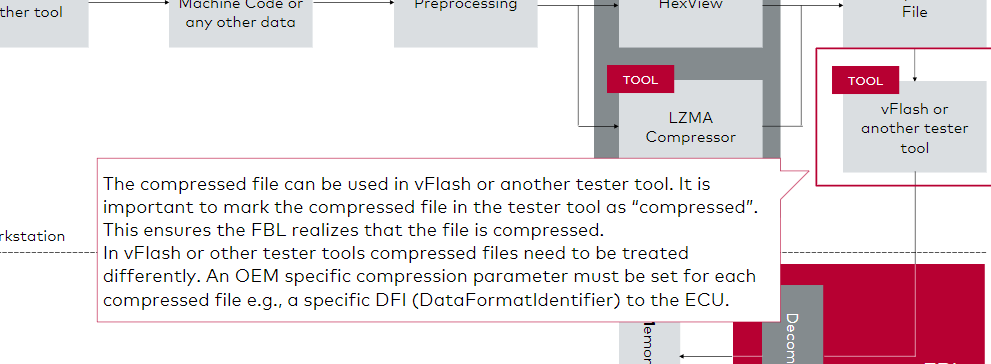
**Procedure of Data Compression/ Decompression**

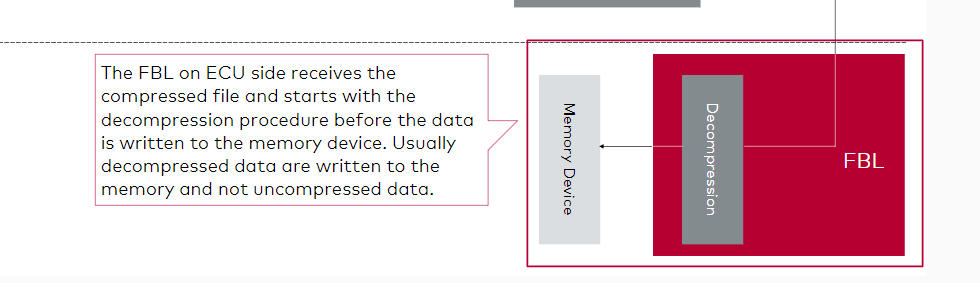
The graphic below describes the compression on the PC side and the transfer of the data to the ECU. On the ECU side, the data is decompressed before it is written to the memory device. Compared to the previous data encryption/decryption chapter the three first steps are equal. Instead of encrypting the data it will be compressed. The “Gap-Filling” or shifting of addresses in hex files can be considered. The resulting data file will be passed to the compressor.









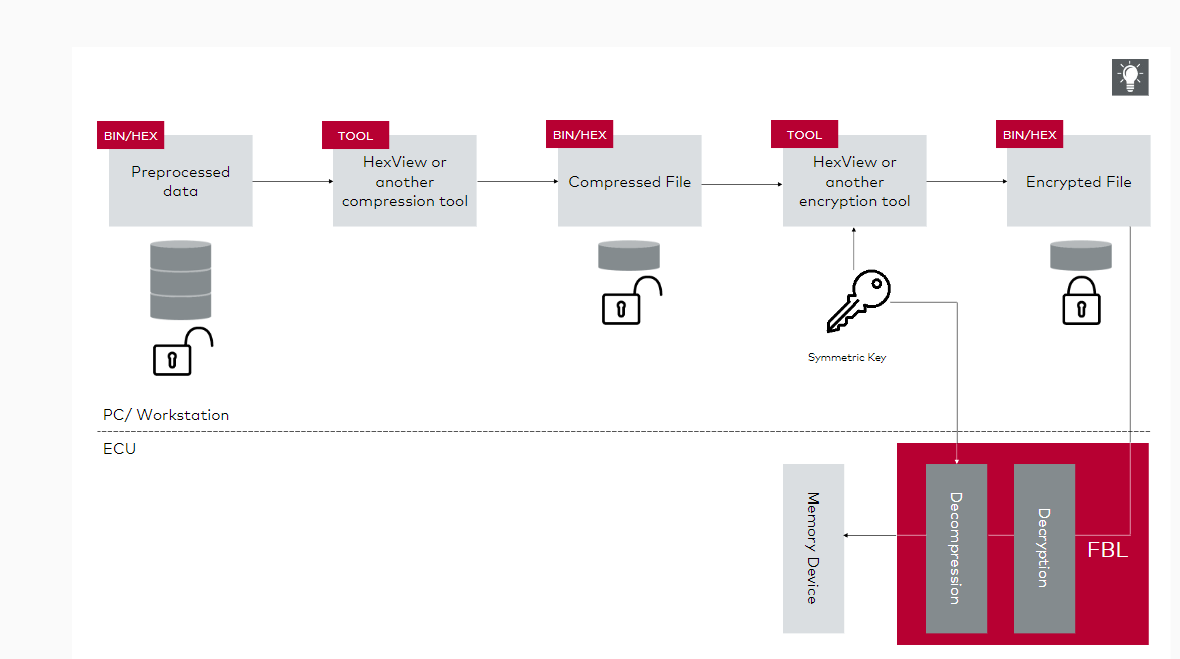


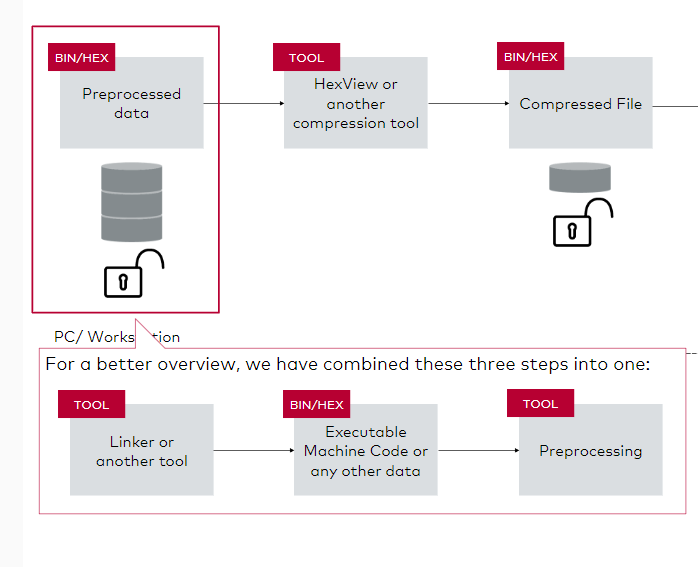
**Caution**

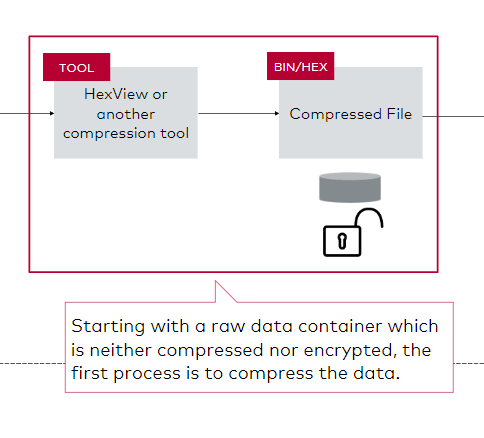
It is also be possible to program the compressed data to the memory. The instance which accesses the compressed data must decompress the data before it can be used. A possible use-case could be that compressed data is stored in a small flash memory. During runtime the data can be decompressed and executed in larger RAM.

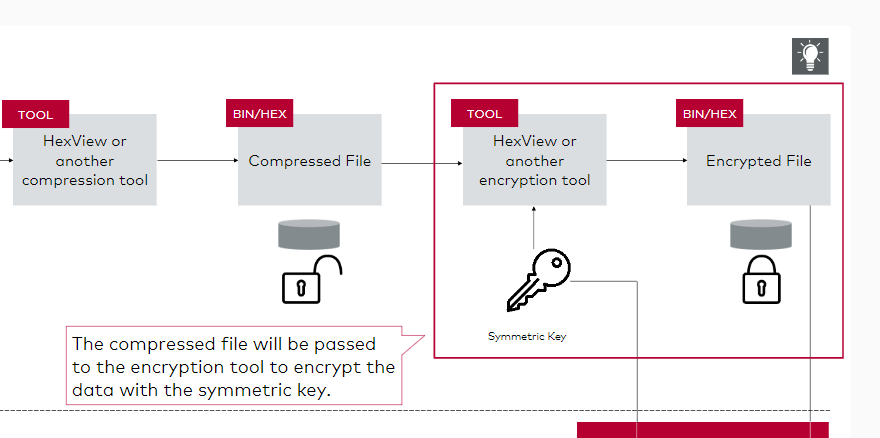
**Combination of Compression and Encryption**

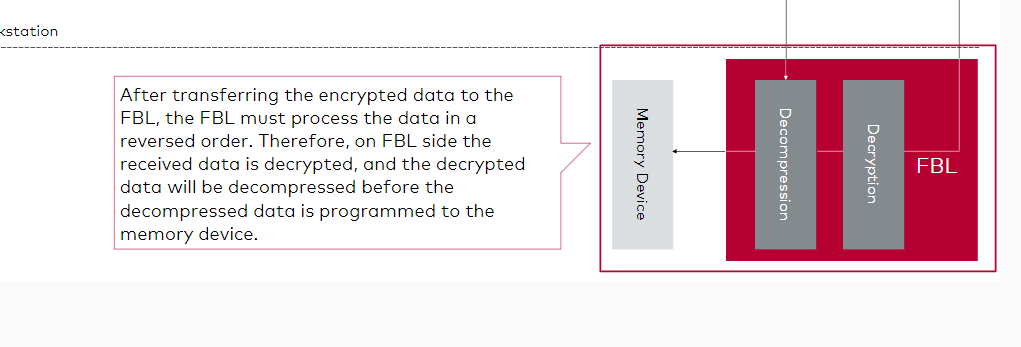
Files for download can be compressed or encrypted. In addition, if compression and encryption is applicable and enabled, a combination of both features is possible. The graphic shows the processing steps and order of the combined features. It is important that processing procedure is complied with the depicted processing order.











**Caution**

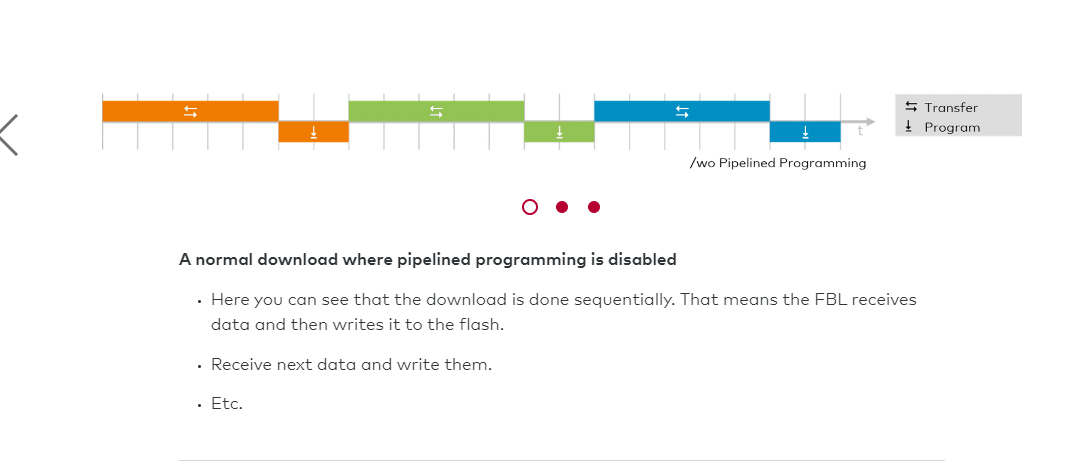
The received encrypted and compressed data is not stored persistently. The data is decrypted and decompressed on-the-fly before the received chunks of data are programmed to the memory device. Having these features enabled on a slow hardware, responses for transfer data requests may take longer compared to the disabled features.

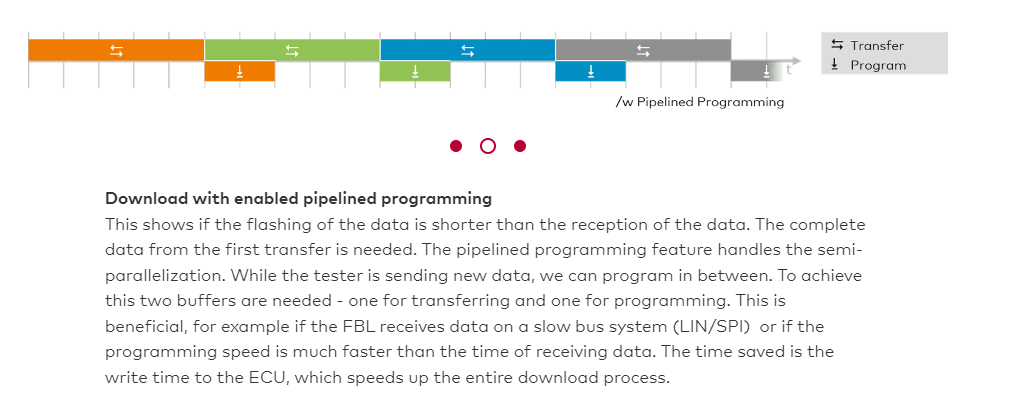
# Pipelined Programming and Verification

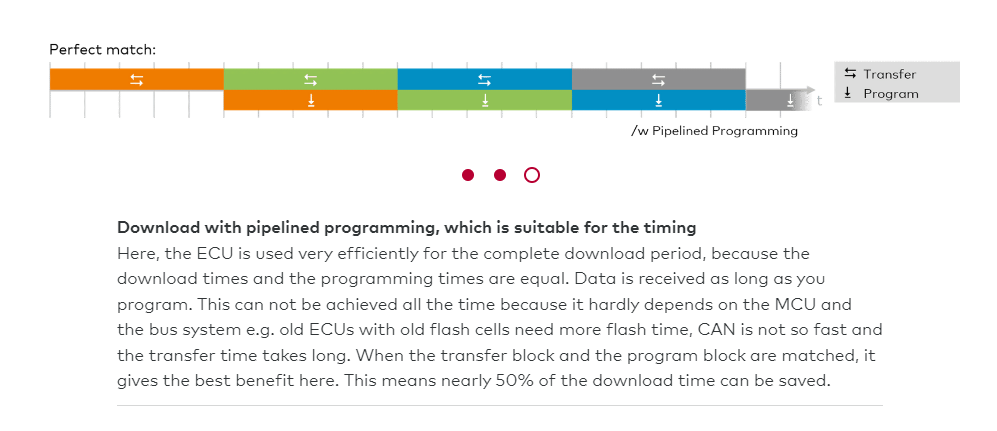
**Pipelined Programming**

Pipelined programming is a method of writing data to memory while receiving new data in parallel.  With this feature a time reduction of up to 50% (typically 20-30%) can be achieved. An additional buffer is required for this. Therefore, you have to consider that the size of the diagnostic buffer in RAM is available twice. Once for the transferred data and once for the data that is written in parallel.

The rest of the download remains the same (erase and verification are not affected).







For the tester it is transparent. Negative response may be delayed because the first is always a positive response.

Sometimes adaptation of OEM specification is required.

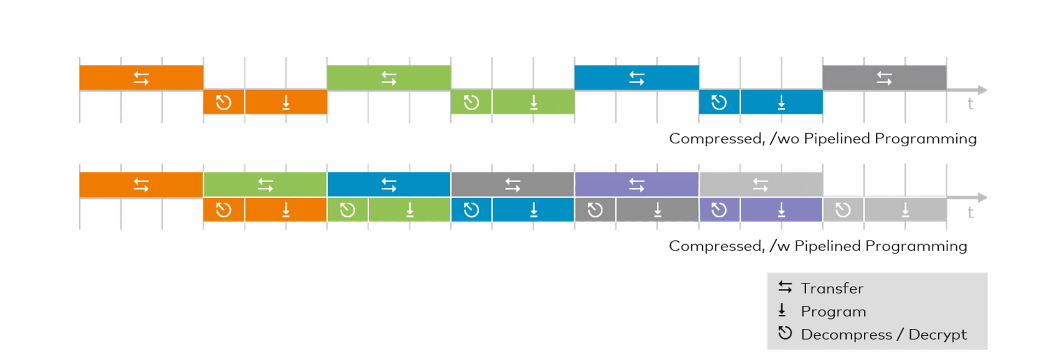
**Caution**

The response handling needs to be mentioned because, if the first block is written and an issue occurs, the Negative Response Code (NRC) is sent after the second block is transferred. So, it looks like the second block fails, even if it's related to the first block.  
Therefore, if there is an error while downloading, the Pipelined Programming feature should be disabled to see where the error really happens.

**Pipelined Decompression and Decryption**

Additionally, decompression and decryption can be pipelined as well. In this case, the decompression must take place before the data is written.

The decompression or decryption takes place in parallel to the download. The spare time during the transfer can be used for programming. Further reduction of download time is possible.



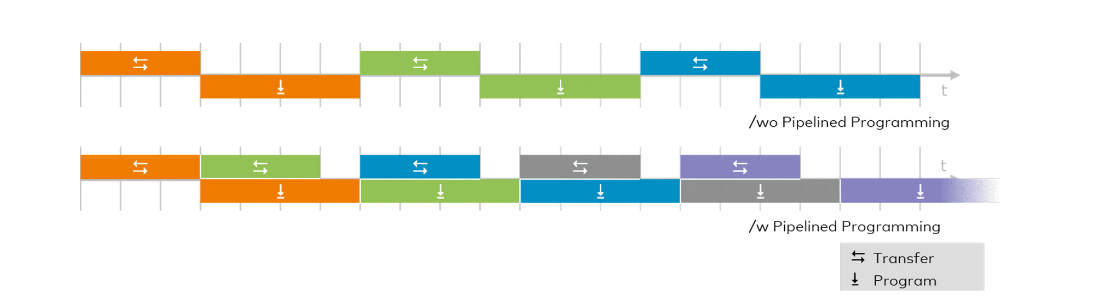
For the processing of two methods (compressed and encrypted) you need also two buffers which are 2x data processing buffer. This could lead to a high RAM consumption, if pipelined programming is enabled. Since for decryption and decompression two diagnostic buffers and two processing buffers are needed:

* one diagnostic buffer for the received data
* one diagnostic buffer for the data which are currently processed
* one processing buffer for decryption
* one processing buffer for decompression

The encrypted and decompressed data will then be written to the memory. There may be limitations due to the respective resources, such as available RAM (buffer, code).

**Request Received while Programming in Progress**

If a request is received while programming is still in progress, the programming is interrupted and the service is processed. The positive response is delayed, and RCR-RP is issued. Programming resumes to free buffers. Pending data from previous request is terminated and current data is programmed in parallel again.  
Interrupting and resuming creates additional overhead. The actual overhead depends on the controller.



If the Flashing takes more time than the transfer (fast bus, slow ECU), the ECU must wait till the flashing has finished before the next transfer will be started.

The improvement depends on the

* download speed (bus dependent),
* data processing performance and
* programming performance (controller dependent).

As a requirement, a lot of RAM resources are needed, because when the flash driver is executed, nothing must be executed from RAM, at least for the internal program flash driver. For pipelined programming we have a lot of things going on at the same time: we are programming and at the same time we are receiving data from the bus system. This means that the communication stack has to be placed in RAM (compiler dependent). The bigger the communication stack, the more RAM resources are needed.

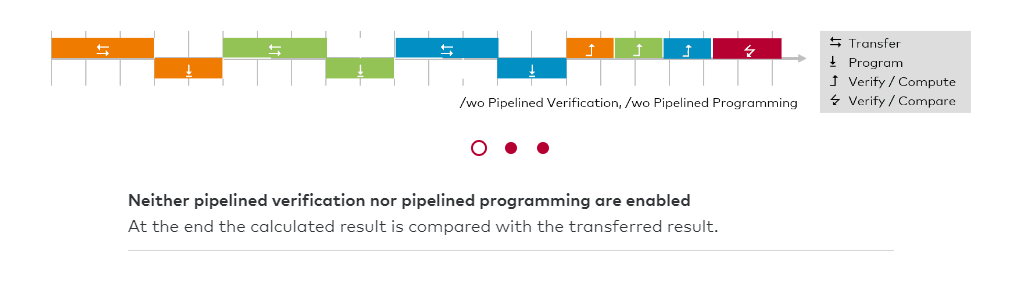
The operation of the flash driver must be interruptible (controller dependent). This means that we should be able to go back at a certain time and fetch the data from the bus system even though we are programming.

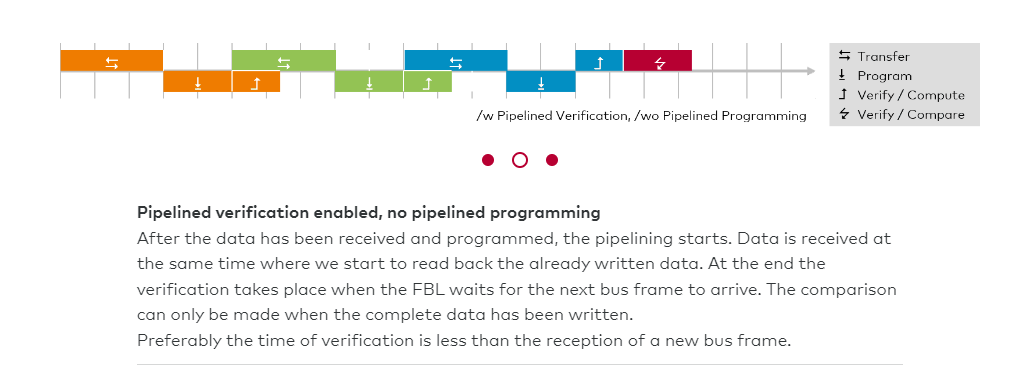
**Note**

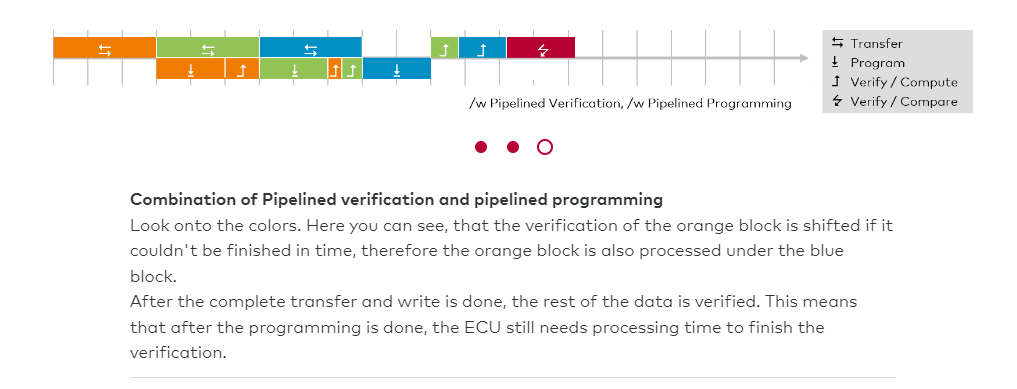
This is not applicable to all platforms. The cost/ benefit ratio depends on the download size and the above factors.

**Pipelined Verification**

On top of pipelined programming, you can use pipelined verification.  
Verification can be done with a checksum or with a signature. To verify data for the complete application the Bootloader has to read back the complete application data, calculate or process the data and compare the result with the transferred result.  
Pipelined Verification enables in the FBL to verify data during the reception of bus frames. It is transparent to the verifier and can also be used for compressed/ encrypted downloads. Pipelined verification can be combined with pipelined programming (see above). It only consumes free processor time. The remaining data is verified at the end of the download.



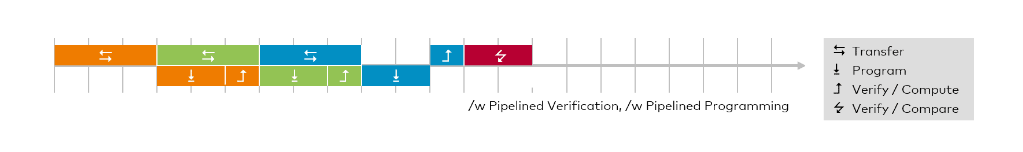




**Tips**

If there are encrypted and compressed data, which must be decrypted and decompressed before it can be programmed, the complete verification step is shifted to the end of the download. The reason is that if the decryption + decompression + programming takes the same time (or more) than the transfer, the verification is shifted to the end of the download like in the first picture.

In an ideal scenario, the programming and verification step can be done in one transfer block. Nevertheless, there is always a final verification step at the end because of the comparison of either the checksum or the signature.



Whether it leads to an improvement depends on the

* download speed (bus dependent),
* programming performance (controller dependent, only relevant in combination with pipelined programming),
* processor speed (controller dependent) and
* complexity of verification algorithm.

# Bootloader Updater

**Bootloader Updater**

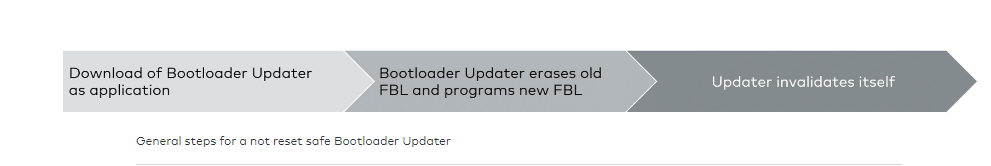
An Bootloader Updater is used for exchanging the FBL via UDS services e.g. if the ECU is already sealed, closed, build in the car or the debug connection is locked, or if updates are necessary in the FBL.

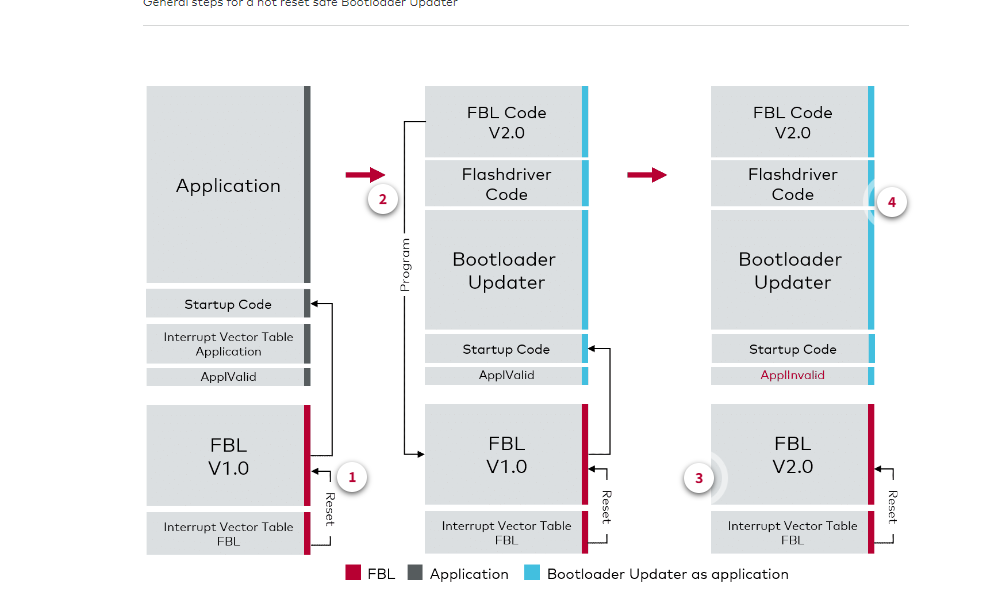
This example is for the non-reset safe updater case. In this case, a failed/aborted updater can cause a dead ECU because

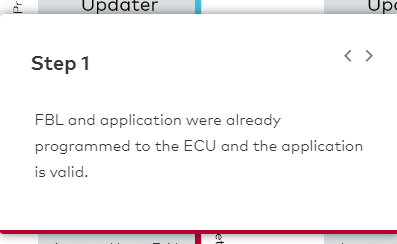
* the FBLis started first,
* the reset vector points to the FBL startup code,
* the location where the FBL is located may be erased,
* voltage drop or programming failure while FBL update,
* there is no possibility to do the update again.

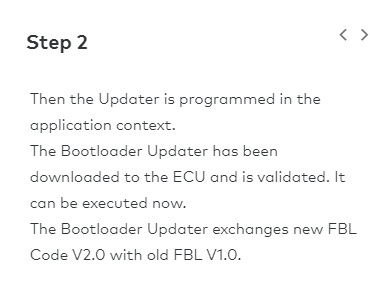
**Non-Reset Safe**

The download of the Bootloader Updater does not distinguish from the normal application download. The reason for this is that the Bootloader Updater is considered as application whose only purpose is to perform the update of the FBL.

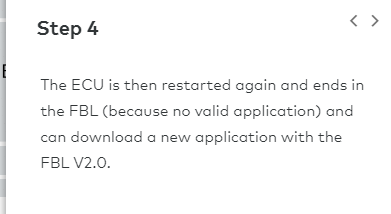












**Tips**

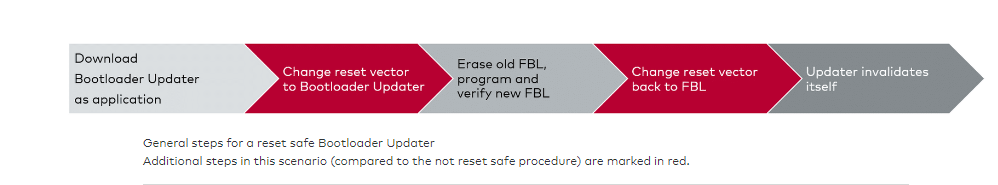
Keep in mind that here the interrupt table is fixed and therefore a reset while updating the FBL V1.0 to FBL V2.0 will cause a dead ECU or a not proper working one, because code is missing. If the FBL is broken, a new application or updater download is not possible anymore.

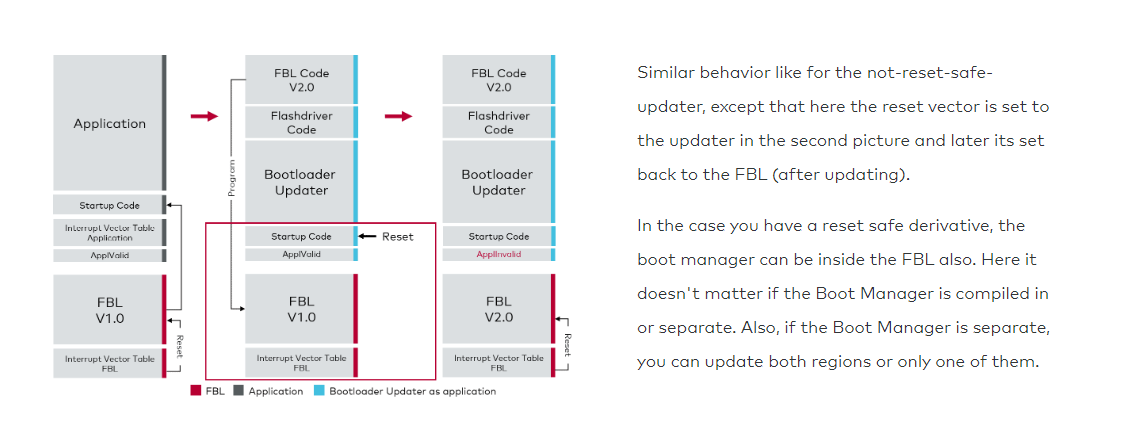
**Reset Safe without Boot Manager**

The update starts from scratch when it is interrupted. After a power loss the update starts again. Not the startup code of the Bootloader will be executed, but the startup code of the Updater. This ensures that always the Updater is started. The reset safeness depends on hardware.

Reset safe updater means that, if a reset occurs (e.g. power cycle), the start address of the software is handled through firmware/reset vector. This can be a register or a flash area where a special pattern is used to determine the start address. Therefore, the updater can relocate this reset vector to start the updater before executing. After an unintended reset, the updater is started again (if it was not finished).

After the updater has finished, it will change the reset vector again to the new Bootloader.



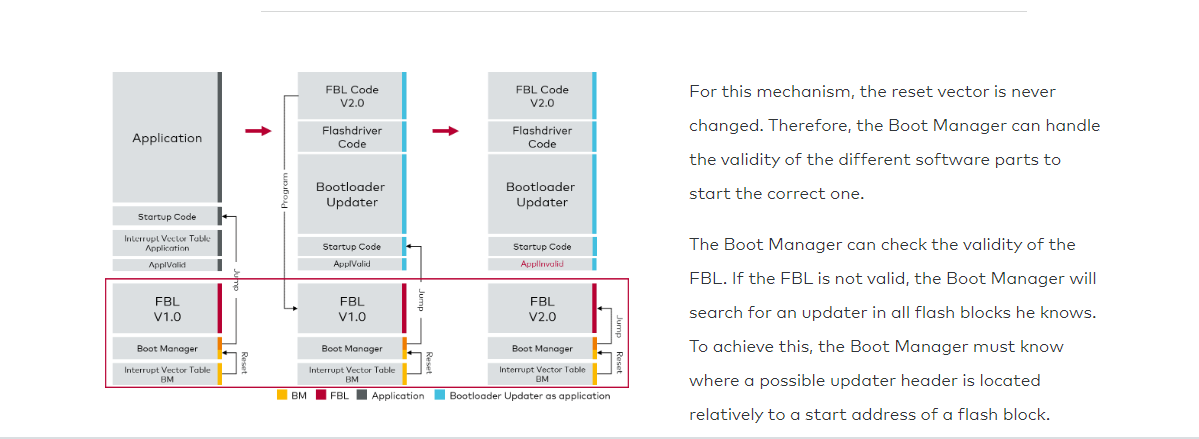
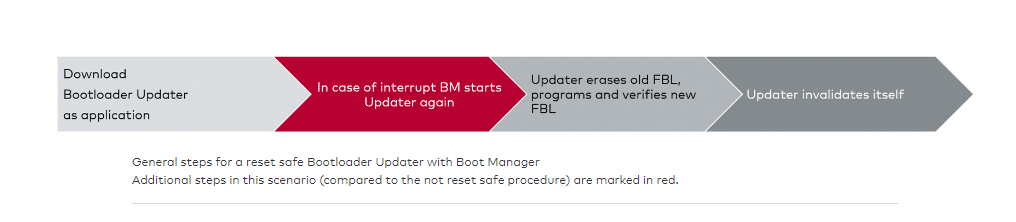


**Reset Safe with Boot Manager**

A method to make a non-reset-safe-updater, to a reset-safe-updater is to use a Boot Manager.

The update again starts from scratch when it is interrupted. Hardware needs no support for reset safeness. Reset vector only points to BM. The Boot Manager is not replaced, only the FBL. BM decides whether application (updater) or FBL is started.

The Boot Manager will check which parts of the software are valid for example an updater or the FBL itself. If the updater resets while executing, the boot manager will realize that the FBL is invalid and will search for an updater to start it up again.



**Example**

The header of the updater is in the 10th flash block with an offset of 0x200. Then the Boot Manager will search over all flash blocks with this offset to find a valid header structure and would start the updater again.

Part 1

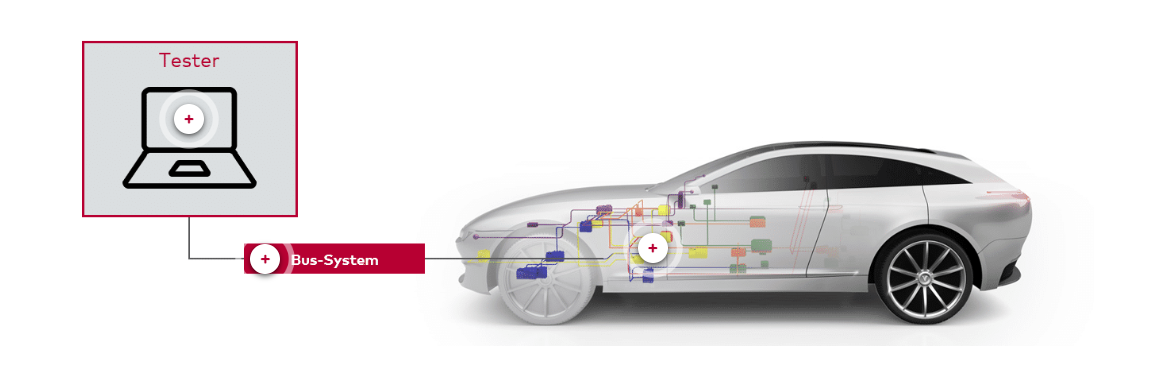
# General Purpose and System Overview

**Overview**

In this chapter you will learn about the general purpose of the Flash Bootloader and get an introduction into the system overview. This includes architecture overview and the memory layout.

**General Purpose of a Flash Bootloader**

Updating an ECU's software even after a car is manufactured became more and more important. The **Flash Bootloader (FBL)** enables the download of new data to ECUs - no matter of the bus system. The ECUs are accessed via a test computer (tester). Click on the highlighted buttons in the graphic below to find out more about the Flash Bootloader and its purpose.



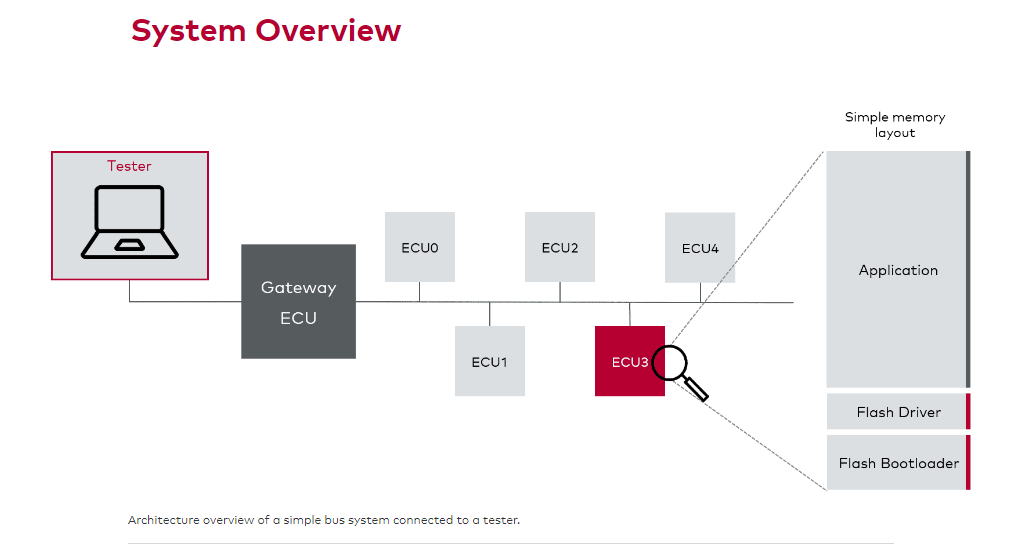
There are **three**common use cases:

* Development: ECU is installed in a prototype vehicle and is not accessible via debug interface
* End-of-line programming: at the end of the car manufacturing the vehicle will receive the latest software updates.
* After-sales/service programming: Software update after purchase (issues, new features)

**What are the advantages of flashing over the bus system?**

–

1. Controller's debugging interface can be disabled for in-series ECUs
2. No hardware replacement necessary because bus system is always accessible
3. There is only one protocol for all ECUs since they are accessed via the tester
4. Improved security e.g., by using signatures on the downloaded data



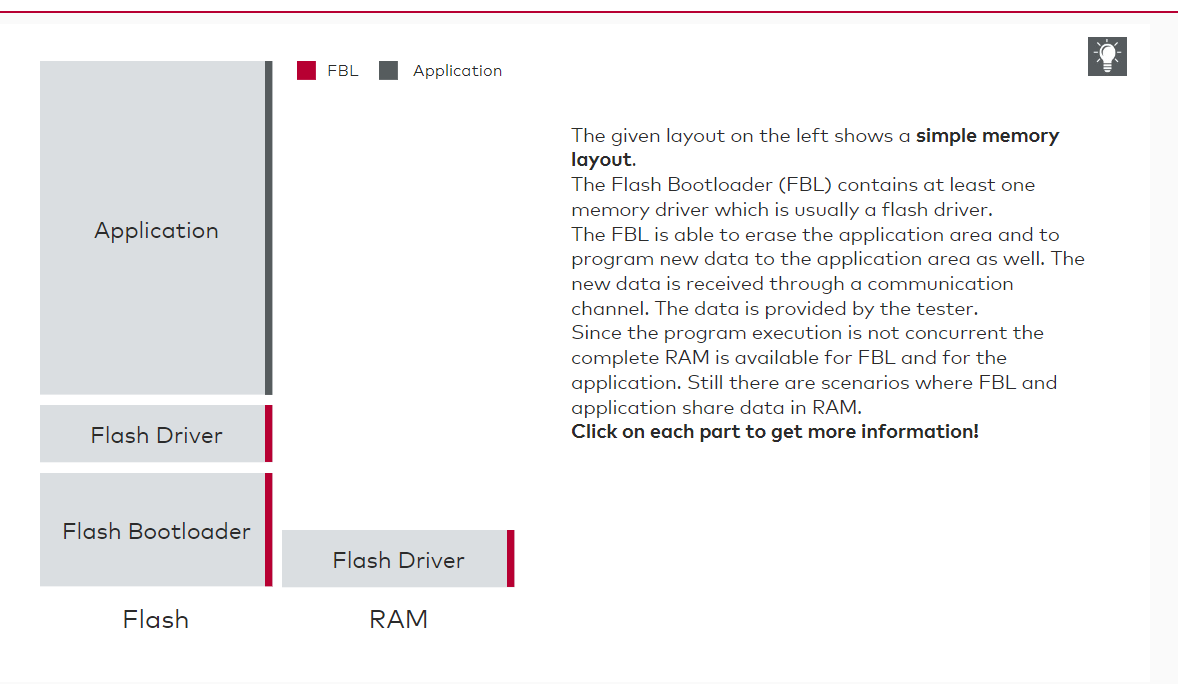
In the graphic above you can see an example of an architecture overview. The tester is connected to a gateway ECU. This gateway ECU is then connected to an in-vehicle bus system. Several ECUs are attached to this bus, but only ECU3 shall be (re)flashed in this example. The ECU3 contains the **simple memory layout**, which consists of a stack of Flash Bootloader, flash driver and the application on top. The memory layout will be explained in more detail just below.

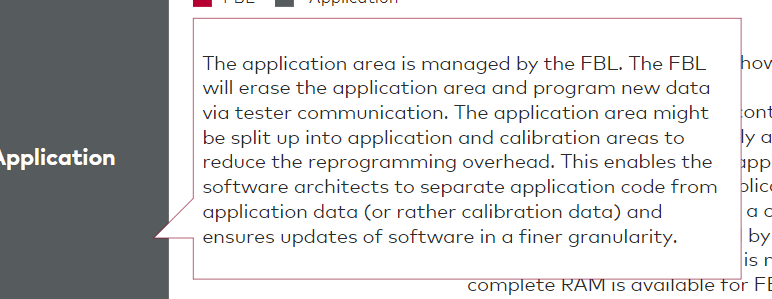
**Memory Layout**

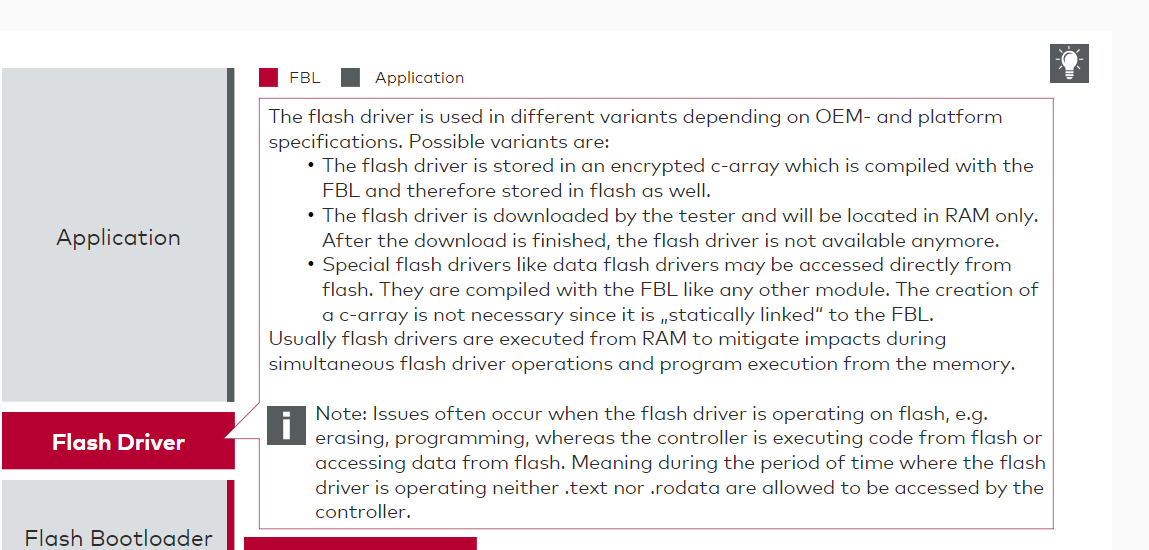
The simplest layout includes the FBL, a flash driver and an application in the internal microcontroller unit's (MCU) flash memory.

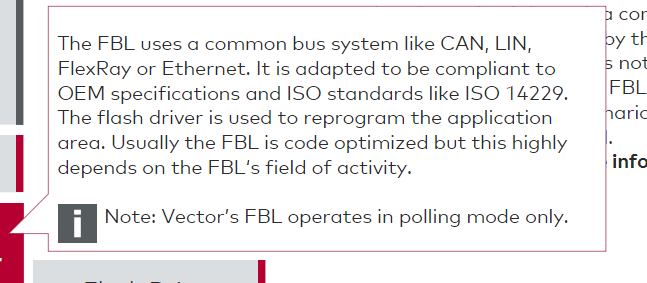
The FBL is placed in the smaller flash blocks of the controller and the application in the larger flash blocks. The reason for this is that the FBL is usually much smaller than the application. Each program (FBL, application, etc.) has to be aligned to the flash blocks of the controller. Therefore, the most reasonable memory layout is placing smaller programs to smaller flash blocks and larger programs to larger flash blocks. The alignment is necessary because flash memory cannot be erased in small chunks but only in larger flash block sizes, like 8KB to 1024KB for common controllers.

The simple memory layout is shown below in the interactive graphic. You can find out more about each part by clicking on them.









**Flash Driver**

You already got to know the flash driver above. But let's take an even closer look at this component. Flash drivers are used to operate on the flash memory. Possible operations are:

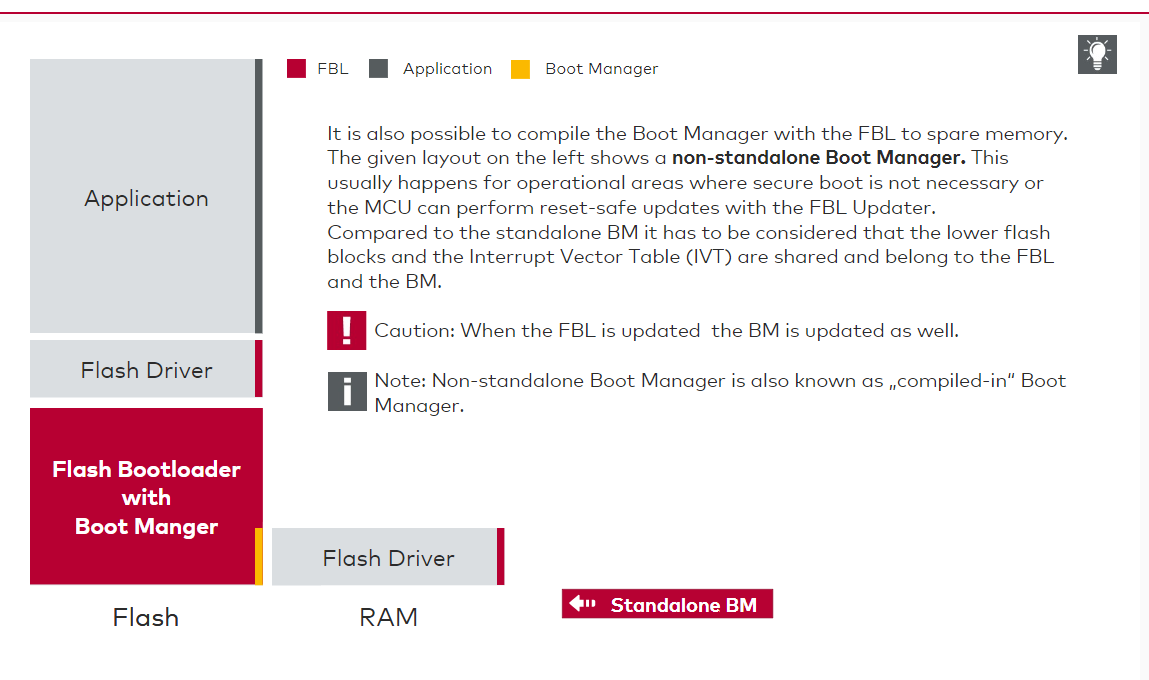
* Erasure of flash blocks
* Programming of data to the memory
* Reading data from memory
* ECC safe reading from memory if applicable

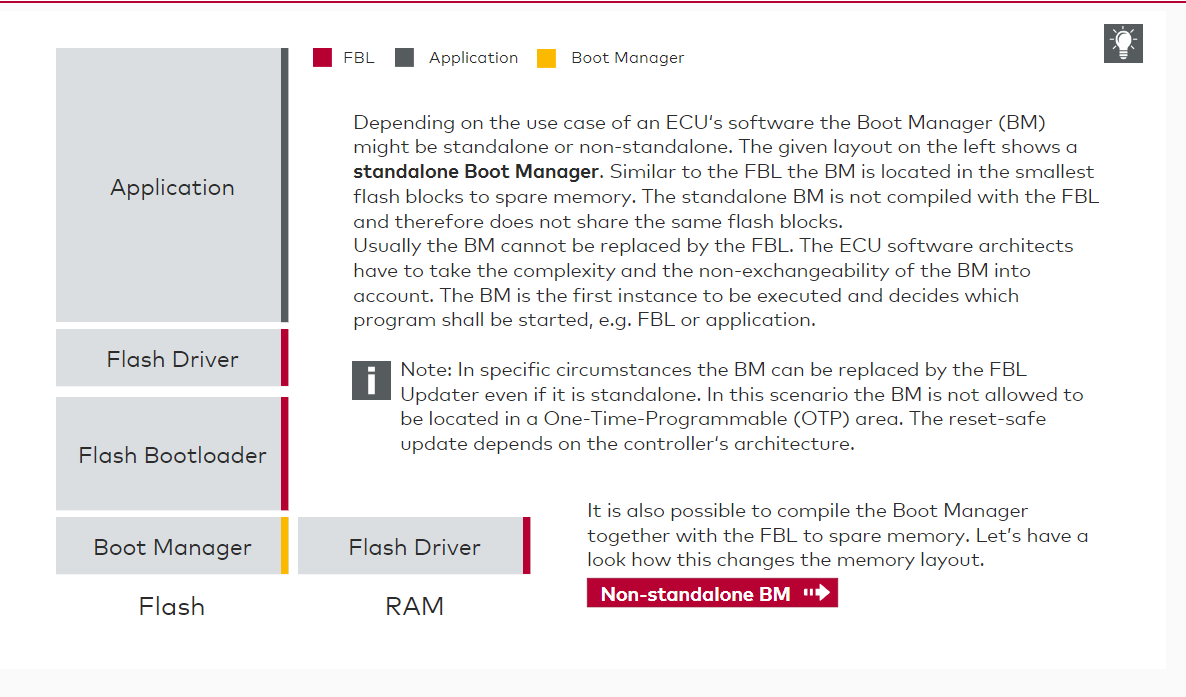
Flash drivers sometimes contain derivative specific routines like adjustments of reset vectors. Most semiconductor vendors of flash memory do not allow writing two times to the same address without erasing the data section. Before reprogramming on an address that already contains programmed data, the complete physical flash block of this address needs to be erased. If this is not considered:

* Wrong data might be persisted
* Exceptions may be thrown by the flash peripheral
* Common data retention time is not ensured, or other issues might occur

**Memory Layout with Boot Manager**

* Next to the simple layout, we should also discuss the layout including a Boot Manager. You can learn the layout with a standalone BM just below. If you click on the button "Non-standalone BM" the graphic will change and you can learn about the non-standalone BM as well.





# FBL in Use and Download Sequence

**Overview**

In this chapter you will get to know how the FBL works when it is actually in use. Furthermore, you will learn about the three phases of the download procedure:

* Pre-programming
* Programming
* Post-programming

Check Notes for the initial sessions on FBL: General purpose Overview

Secure boot

fota